WCF 4

WINDOWS COMMUNICATION FOUNDATION И .NET 4

ДЛЯ ПРОФЕССИОНАЛОВ

PROFESSIONAL

WCF 4

WINDOWS COMMUNICATION FOUNDATION WITH .NET 4

Pablo Cibraro
Kurt Claeys
Fabio Cozzolino
Johan Grabner

Wiley Publishing, Inc.

WCF 4

WINDOWS COMMUNICATION FOUNDATION И .NET 4

ДЛЯ ПРОФЕССИОНАЛОВ

Пабло Сибраро
Курт Клайс
Фабио Коссолино
Йохан Грабнер

Диалектика

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
С34
УДК 681.3.07

Компьютерное издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского докт. физ.-мат. наук *Д.А. Клюшина*, *В.А. Коваленко*,
канд. техн. наук. *И.В. Красикова*, *Я.К. Шмидского*

Под редакцией *В.А. Коваленко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Сибраро, Пабло, **Клайс**, Курт, **Коссолино**, Фабио, **Грабнер**, Йохан.
С34 WCF 4: Windows Communication Foundation и .NET 4 для профессионалов. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 464 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1713-3 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Пабло Сибраро, Курт Клайс, Фабио Коссолино, Йохан Грабнер

WCF 4: WINDOWS COMMUNICATION FOUNDATION И .NET 4
ДЛЯ ПРОФЕССИОНАЛОВ

Литературный редактор *И.А. Попова*
Верстка *Л.В. Чернокозинская*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 28.12.2010. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 29,0. Уч.-изд. л. 25,4.
Тираж 1500 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1713-3 (рус.)
ISBN 978-0-470-56314-4 (англ.)

Оглавление

Содержание

Посвящаю эту книгу моей жене Ромине и маленькой дочери Джезмин. Вы – истинная радость в моей жизни.

Пабло Сибраро

Я хотел бы поблагодарить мою жену Мэриджк за то, что она позволяла мне сидеть ночи напролет за компьютером, а также моих детей, Милана и Тимона, дающих в перерывах от написания книги так необходимое мне удовольствие пребывания вместе с ними.

Курт Клайс

Тициана, твоя поддержка важна для меня.

Фабио Коссолино

Посвящаю эту книгу своей семье, племяннице Сэскии, племяннику Алексею и всем моим друзьям. Спасибо.

Йохан Грабнер

Об авторах

Пабло Сибраро — региональный CTO в Tellago Inc. и всемирно признанный эксперт с более чем десятилетним опытом в области архитектуры и реализации больших распределенных систем, в которых применяются технологии Microsoft. Несколько лет, работая непосредственно в различных командах Microsoft, он разрабатывал примеры приложений и занимался написанием руководств по созданию приложений для обслуживания широкого круга запросов с помощью веб-служб, расширения веб-служб (Web Services Enhancements — WSE) и Windows Communication Foundation (WCF). Он также занимался технологиями, которые позволяют разработчикам создавать крупномасштабные системы, такие как WCF, WF, Dublin, OSLO и Windows Azure.

Курт Клайс — архитектор решений .NET и тренер по разработке программного обеспечения во многих областях с более чем пятнадцатилетним практическим опытом. Он родился и работает в Бельгии. В настоящее время его внимание сконцентрировано на архитектуре для обслуживания широкого круга запросов и интеграции прикладных систем уровня предприятия на основе Windows Communication Foundation, BizTalk и Azure AppFabric. После работы в VB и ASP (с 1995 года) в 2002 году он начал работать с .NET, а в 2005 году был очарован WCF. Сейчас Курт работает в нескольких финансовых компаниях и правительственных институтах в Брюсселе в качестве архитектора, ведущего разработчика и тренера. Имея титулы MCT и MCSD, он занимается обучением .NET 3.5/4.0 и Azure; в TechEd North America и TechEd Europe выполняет обязанности преподавателя Technical Learning Guide и Instructor Led Lab. Его страсть к технологии также выражается в выступлениях на многочисленных мероприятиях сообщества и Microsoft во всем мире в области WCF и Azure. В 2008 году за общественно-полезный труд он был награжден званием системного разработчика MVP Connected System Developer. Кроме того, Курт — компетентный лидер в подразделении 150 разработчиков .NET в ORDINA (Бельгия), где он отвечает за наставничество сотрудников, применяющих в своей работе WCF, BizTalk и Azure AppFabric. Если он не запускает Visual Studio 2010 на своем PC и не находится в классной комнате, то любит проводить время дома с женой и детьми или совершает кругосветное путешествие. Связаться с Куртом можно по адресу `www.devitect.net`.

Фабио Коссолино — архитектор программного обеспечения, аналитик и разработчик, в настоящее время работающий в FimeSan, CompuGroup Company. Особую страсть он питает к развитию электронных медицинских платформ и решений. Он часто выступает спикером на мероприятиях и встречах групп пользователей на юге Италии. Как правило, темы его докладов связаны с WCF и Windows Azure. Он также активно работает в сообществе как лидер DotNetSide, итальянской группы пользователей .NET. Фабио пишет многочисленные статьи для самого популярного итальянского журнала по программированию и издал на итальянском языке руководство *Windows Communication Foundation*, основанное на финальной версии WCF. За его длительную и высококачественную работу в сообществах в 2010 году он был награжден MVP в категории системных разработчиков (Connected System Developer). В 2004 году Фабио получил сертификат MCAD (Microsoft Certified Application Developer) по .NET. У него есть два блога: `http://dotnetside.org/blogs/fabio` (итальянский язык) и `http://weblogs.asp.net/fabio` (английский язык). Вы можете также найти Фабио на твиттере `http://twitter.com/fabiocozzolino`.

Йохан Грабнер живет в Граце, Австрия. Он занимается разработкой программного обеспечения и баз данных почти пятнадцать лет. Будучи студентом экономического факультета, в 1991 году он начал разрабатывать приложения баз данных. А в 1997 году с энтузиазмом начал работать внештатным сертифицированным тренером Microsoft по разработке программного обеспечения и баз данных, получив все необходимые сертификаты, такие как MCPD, MCITP, MCDBA, MCAD, MCSD, MCSD.Net, MCT, MCSE, OCP и SCJP. Йохан выполнил несколько больших разработок программного обеспечения и образовательных проектов для австрийских и международных компаний в различных областях бизнеса, используя технологии Microsoft .NET. Он консультирует, тренирует и реализует службы в области разработки программного обеспечения для обслуживания широкого круга запросов и программирования на основе SharePoint и баз данных. Йохан имеет звание системного разработчика MVP Connected System Developer с 2008 года. Он был спикером и техническим руководителем на различных конференциях, включая Austria.NET, TechEd North America и TechEd Europe, в дополнение к тому, что является инспектором (Proctor) ASP.NET и VB.NET и презентатором интернет-трансляций в нескольких европейских странах. Йохан также регулярный спикер в .NET Usergroup South Austria.

Признательности

Я хотел бы поблагодарить свою жену Ромину за ее терпение, которое она проявила в то время, когда я работал над этой книгой. Также особая благодарность моему хорошему другу и Tellago CTO, Иисусу Родригесу, вдохновение которого и поддержка от начальной буквы до последней стадии позволили мне написать эту книгу. Наконец, искренне благодарю сотрудников Wiley и Курта Клайса за то, что дали мне возможность написать книгу по моему любимому предмету — WCF.

Пабло Сибраро

Хочу поблагодарить всех людей, которые помогали мне в написании этой книги, — без их поддержки она не появилась бы. Хочу поблагодарить свою семью, жену Мэриджк за то, что позволила мне ночами напролет сидеть за компьютером, и моих детей, Милана и Тимона, за то, что в перерывах от создания книги доставляли так необходимое мне удовольствие побыть вместе с ними. Вы делаете мою жизнь радостью. Хотя дети все еще маленькие, я чувствую, что они знали то, что я делал все время: упорная работа, направленная на реализацию мечты. Я также хочу поблагодарить моих родителей за то, что они поверили в меня несколько лет назад, когда я решил заняться компьютерами и изучить информатику. Особая благодарность Роберту (Бобу) Эллиоту (Robert (Bob) Elliott) за то, что дал мне возможность написать эту книгу и сформировал коллектив. Он — великий человек и может действительно мотивировать людей в бурные времена. Я также хочу поблагодарить своих сотрудников в Ordina; они очень помогли мне и поддержали меня своими откликами. Конечно, в самом большом долгу я перед моими соавторами: Пабло Сибраро, Фабио Коссолино и Йоханом Грабнером. Именно они сделали всю тяжелую работу в этой книге. Наконец, хочу поблагодарить всех великих людей, работающих в Microsoft Belgium, за их дружбу.

Курт Клайс

Хочу в первую очередь поблагодарить других трех авторов, Курта, Пабло и Йохана, которые позволили мне работать вместе с ними. Особая благодарность Пабло, который верил в меня и в мою работу. Спасибо также команде из группы пользователей DotNetSide (DotNetSide User Group), Тициане, Вито, Марио, Лео и Андреа, которые всегда поощряют меня в каждом из моих начинаний в сообществе, и всем моим коллегам, особенно Алессандро, Бенедетто, Миммо и Винченцо, которые оказывают мне большую моральную поддержку. Также большое спасибо Клаудио, моему менеджеру по научным исследованиям и развитию (Research & Development Manager), за его вдохновение, поддержку и внимание к моей работе.

Большое спасибо сотрудникам Wiley. Роберту Эллиоту (Robert Elliot) и Келли Тэлбот (Kelly Talbot), с которыми я вел огромный обмен почтой по организации и координации моей работы. Спасибо Трисии Либиг (Tricia Liebig), Джеффу Райли (Jeff Riley) и Дугу Холлэнду (Doug Holland) за их тяжелую работу по рассмотрению содержания глав. Их отклик был очень важен для успеха этой книги.

Работа над этой книгой потребовала потратить несколько бессонных ночей на написание текста, и я благодарю моих родителей и мою сестру за их терпение.

Наконец, я хотел бы поблагодарить свою жену Тициану за ее вдохновение и поддержку в работе над этой книгой во время подготовки к нашей свадьбе. Слова не могут выразить то, что я чувствую. Без нее я не мог бы сделать все это, и моя жизнь никогда не была бы такой счастливой.

Фабио Коссолино

Прежде всего хочу поблагодарить мою семью и друзей за их сочувствие, а также за огромную поддержку, доверие и вдохновение, которое они дали мне. Если бы не приятные часы и дни, когда моя семья баловала меня хорошей едой, и не мои друзья, которые приходили с хорошим пивом, чтобы восстановить мою энергию, боюсь, что бросил бы писать и работать над главами через несколько недель.

Будучи соучредителем .NET Usergroup South Austria, я должен также поблагодарить членов сообщества и коллег из Microsoft за их конструктивные предложения и оживленные обсуждения.

Должен также выразить благодарность Келли и Бобу, которые следили за тем, чтобы работа продвигалась по графику, и, конечно же, Курта Клайса и Пабло Сибраро за их превосходное сотрудничество.

Надеюсь, эта книга содержит много полезной информации, которую читатели смогут применить в своей работе. Эта информация должна служить идеальной отправной точкой всестороннего изучения WCF.

Введение

В этой книге рассказывается о Windows Communication Foundation в .NET 4.0. WCF является технологией в .NET, которая используется для создания приложений, предназначенных для широкого круга запросов, обмена сообщениями в различных коммуникационных сценариях и выполнения потоков работ, состоящих из действий служб. Эта новая книга поможет понять принцип ориентации на службы, изучить примеры коммуникации и научиться декларативно описывать бизнес-процессы. Вы изучите различные части технологии, предназначенные для поддержки таких сценариев, и получите четкое представление о том, как части WCF 4.0, работая вместе, служат основой поддержки многих аспектов распределенных приложений уровня предприятия. Помимо объяснения технических аспектов стека WCF 4.0, в книге объясняется также практика разработки, выделяются три случая (ориентированность на службы, передача и бизнес-процессы) и их реализация шаг за шагом. Авторы научат разрабатывать приложения для WCF с помощью Visual Studio.

При создании своих приложений на платформе WCF 4.0 вы также научитесь эффективно использовать Visual Studio 2010 для создания решений, в которых максимально применяются новые возможности WCF 4.0.

В данной книге описывается, как взаимодействуют разработчик и архитектор при создании приложений, которые интегрируются в новую парадигму программирования в WCF 4.0. Вы также узнаете, как установить решение в этой новой архитектурной форме, основанной на WCF 4.0, с помощью технологий и служб .NET. Научитесь устранять реальные проблемы, которые могли бы возникнуть при реализации WCF/WF 3.5 и новой парадигмы программирования, а также новых архитектурных стилей, необходимых для понимания проектов WCF 4.0. Примеры, приведенные в книге, идут гораздо глубже примеров “hello world”, ведут к архитектурно корректным решениям и служат руководствами по наилучшим методам программирования.

Авторы имеют огромный опыт в реализации технологии в реальных проектах. Они ежедневно сталкиваются с такого рода задачами, находят наилучшие их решения, объединяя лучшие методы и руководства, и применяют свой опыт на практике.

Для кого предназначена книга

Эта книга предназначена для разработчиков .NET средней квалификации и архитекторов решений, которые заинтересованы в использовании WCF 4.0 для создания приложений, обслуживающих широкий круг запросов, реализации коммуникации, бизнес-процессов, а также безопасной и масштабируемой интеграции в “облака”.

Что рассматривается в книге

- Разработка служб и использование передачи и модели расписания работ в корректной и надежной архитектуре.
- Различные реализации связи в WCF.
- Преимущества улучшенного обмена сообщениями в WCF 4.0.
- Как инстанцировать службы и работать с прокси.
- Различные способы защиты доступа к действиям служб.
- Реализация WCF при служебно-ориентированном подходе.

- Работа со службами расписания действий для организации бизнес-процессов и создания длительно выполняющихся действий.
- Интегрирование своего приложения в "облако" с помощью служб .NET.
- Создание служб REST и использование их с "легкими" клиентами.

Структура книги

В зависимости от своих знаний WCF 4.0 на данный момент можете начать читать любую главу, но мы советуем читать их последовательно, если вы плохо знакомы с данной технологией. Так или иначе, в главе 2 приводится пример реализации проката автомобилей. Это — основа для многих примеров из этой книги, поэтому было бы неплохо ознакомиться с главой 2 прежде, чем читать другие главы.

В главе 1 описываются многие принципы и модели, ориентированные на службы, интеграцию и бизнес-процессы, и показано, как они связаны с WCF. Кроме того, речь пойдет о том, как использовать WCF для реализации моделей. Основное внимание в главе уделяется архитектурному фону в тех областях, где используется WCF.

Главы 2–10 посвящены непосредственно технологии, а также API, предназначенным для разработки приложений, и способам их настройки. В этих главах обсуждаются различные аспекты стека WCF: соединения, клиенты, инстанцирование, расписание действий, безопасность, а также службы .NET. Эти главы предоставят разработчику знания, необходимые для начала программирования в WCF 4.0.

В главах 11–13 шаг за шагом показано, как полностью реализовать решение в Visual Studio 2010. Эти главы реализуют более практический подход и описывают методы разработки полных решений на основе знаний, почерпнутых из глав 1–10. В этих главах вы должны работать с Visual Studio, чтобы завершить проект как решение для данного сценария. Код для этих решений также доступен по адресу `www.wrox.com`.

Каждая из этих глав имеет дело с одним случаем.

- Случай SOA (глава 11).
- Случай коммуникации и интеграции (глава 12).
- Случай бизнес-процесса (глава 13).

Последняя глава этой книги посвящена хостингу. В ней рассматривается хостинг на IIS/WAS и в "облаках". Здесь также обсуждается, как отследить конечные точки и управлять ими с помощью Windows Server AppFabric. В главе также объясняется служба маршрутизации.

Что понадобится при чтении этой книги

Вам понадобится Visual Studio 2010 Professional и .NET 4.0, чтобы изучить WCF 4 и выполнить все примеры, приведенные в данной книге. Можно запустить Visual Studio 2010 на Windows XP Service Pack 3 (кроме Starter Edition), Windows Vista Service Pack 1 (кроме Starter Edition), Windows 7, Windows Server 2003 SP2, Windows Server 2003 R2, Windows Server 2008 SP2, Windows Server 2008 R2. Оперативная память вашей машины должна быть емкостью по крайней мере 1024 Мбайт, но предпочтительно больше.

Соглашения, принятые в книге

Чтобы помочь при чтении текста отследить, что происходит, в книге используются следующие соглашения.

Содержится важная информация, которая непосредственно относится к окружающему тексту.

Указывает на примечания, подсказки или обсуждаемые приемы.

В тексте используются такие стили.

- Мы выделяем новые термины и важные слова, когда вводим их.
- Мы показываем нажатия клавиш на клавиатуре вот так: <Ctrl+A>.
- Код представлен двумя способами.

```
Мы используем моноширинный шрифт без выделения для большинства примеров кода.
Мы используем полужирный шрифт, чтобы подчеркнуть код, который особенно важен в
обсуждаемом контексте, или показать изменения в предыдущем фрагменте кода.
```

Исходный код

Прорабатывая примеры из этой книги, вы можете ввести весь код вручную или использовать файлы исходных кодов, которые сопровождают книгу. Все исходные коды из этой книги доступны по адресу `http://www.wrox.com`. На сайте введите заголовок книги (используйте поле поиска Search или один из заголовков в списке) и щелкните на ссылке загрузки кода Download Code, чтобы получить весь исходный код для книги. Код, который имеется на веб-сайте, выделяется значком, показанным ниже.

Доступно для загрузки на Wrox.com

Листинги содержат имя файла в заголовках. Если это будет только фрагмент кода, то вы увидите имя файла в примечании к коду, например, таком.

Фрагмент кода имя файла

Поскольку у многих книг есть подобные заголовки, самый легкий способ найти нужную, — запомнить ISBN (ISBN этой книги 978-0-470-56314-4).

Как только загрузите код, распакуйте его вашим любимым инструментом сжатия. Или же можно зайти на основную страницу загрузки кода Wrox по адресу `http://www.wrox.com/dynamic/books/download.aspx`, чтобы увидеть код, доступный для этой книги и всех других книг издательства Wrox.

Опечатки

Мы прилагаем все усилия, чтобы избежать ошибок в тексте или коде. Однако ничто не совершенно, и ошибки действительно происходят. Если вы найдете ошибку в одной из наших книг, например ошибку правописания или дефектную часть кода, и сообщите о ней, мы будем очень благодарны вам. Сообщая об опечатках, вы поможете другим

читателям сэкономить время и одновременно поможете нам еще более повысить качество информации.

Чтобы найти страницу опечаток для этой книги, зайдите на http://www.wrox.com и определите местоположение заголовка, используя поле поиска Search или один из списков заголовков. Затем на нужной странице щелкните на ссылке опечаток Book Errata. На этой странице можно просмотреть все опечатки, которые были представлены для этой книги и отправлены редакторам Wrox.

Полный список книг, включая ссылки на опечатки в каждой из них, также доступен на www.wrox.com/misc-pages/booklist.shtml.

Если вы не найдете "свою" ошибку на странице опечаток, зайдите на www.wrox.com/contact/techsupport.shtml и заполните форму, чтобы отправить нам найденную вами ошибку. Мы проверим предоставленную вами информацию и, если она подтвердится, добавим объявление к странице опечаток книги и устраним проблему в последующих выпусках книги.

P2P.WROX.COM

Если хотите написать книгу или вести равноправное обсуждение, присоединитесь к форумам P2P на p2p.wrox.com. Форумы в сети предназначены для объявлений, касающихся книг Wrox, и связанных технологий и взаимодействия с другими читателями и пользователями, использующими описываемые технологии. Форумы предлагают функцию подписки, что позволит посылать вам по электронной почте интересующие вас темы по вашему выбору тогда, когда новые сообщения появляются на форумах. В этих форумах принимают участие авторы Wrox, редакторы, другие эксперты в отрасли, а также читатели.

На http://p2p.wrox.com вы найдете много различных форумов, которые помогут вам не только при чтении этой книги, но и при разработке ваших собственных приложений. Чтобы присоединиться к форумам, выполните следующее.

1. Зайдите на p2p.wrox.com и щелкните на ссылке Register.
2. Ознакомьтесь с условиями использования и щелкните на Agree.
3. Чтобы присоединиться, укажите запрошенную информацию (по желанию можете добавить также любую дополнительную информацию) и щелкните на Submit.
4. Вы получите электронное письмо, в котором будет указано, как проверить вашу учетную запись и завершить процесс присоединения.

Можно также читать сообщения на форумах, не присоединяясь к P2P, но, чтобы отправить ваши собственные сообщения, следует присоединиться.

Как только вы присоединитесь, можно разместить новые объявления и ответить на объявления других пользователей. В сети сообщения можно читать в любое время. Если хотите, чтобы по электронной почте вам отправляли новые сообщения от определенного форума, щелкните на пиктограмме Subscribe to this Forum, расположенной рядом с именем форума в списке форумов.

Чтобы получить дополнительную информацию о том, как использовать Wrox P2P, убедитесь, что прочитали FAQ P2P и знаете, как работает программное обеспечение форума, а также ответы на общие вопросы, относящиеся к P2P и книгам издательства Wrox. Чтобы прочитать FAQ, щелкните на ссылке FAQ на любой странице P2P.

1

Принципы и модели проектирования

В ЭТОЙ ГЛАВЕ...

- Обзор служб и SOA
- Модели коммуникации и интеграции
- Работа с моделями бизнес процессов

В этой главе описывается ряд принципов и моделей служебной ориентации, интеграции и бизнес-процессы. Вы узнаете, как эти принципы соотносятся с WCF и как использовать WCF для реализации этой модели.

Что такое SOA

SOA, или служебно-ориентированная архитектура, — стиль программирования и архитектурный подход к разработке программного обеспечения, при котором приложение организовано в функциональные элементы кода с заданным поведением, называемые службами.

Служба — это группа методов с общим набором требований и общими функциональными целями. Служба вызывается другими модулями, которым нужно выполнить соответствующие действия в зависимости от результата (как, например, данные, результаты вычислений и т.д.). Функции имеют явно определенную и общедоступную сигнатуру, которая публикуется таким образом, что другая программа (клиент службы) может пользоваться функциями службы как черным ящиком. Служебные действия невидимые, — нет никакого прямого взаимодействия с пользователем, и работа выполняется как инструкция, заданная входными параметрами. SOA позволяет упорядочивать

распространяемые приложения. Это означает, что клиенты службы и сами службы запускаются на различных машинах. Такой подход позволяет централизовать бизнес-логику и пользовательский интерфейс или децентрализовать работу других пользователей с помощью сети. Чтобы это было возможным с помощью SOA, машины, хранящие данные, посылают сообщения определенной структуры.

Главная идея SOA — создать нежестко связанную систему, в которой клиент службы и реализация службы имеют в общем случае список доступных для всех служебных действий и определенную структуру параметров.

Клиент знает только сигнатуры, описывающие имя службы, имена и виды входных параметров и тип, возвращаемый функцией службы. Никакой другой зависимости нет. Прикладные платформы и языки программирования могут быть различны как у клиента, так и у службы.

Совершенно ясно, что эта зависимость — исключительно функциональная и не зависит от технической инфраструктуры. Это позволяет легко организовать взаимодействие приложений на различных платформах со службой. Технический краеугольный камень в парадигме SOA — использование стандарта SOAP. (SOAP — язык XML, определяющий содержимое сообщений, которые принимаются и обрабатываются функциями службы.)

Сообщения формируются вне значения параметров или возвращаемого значения, а данные форматирует SOAP. Каждая платформа разработки имеет стек SOAP, так что работа со службой поддерживается во многих средах разработки. Поддержка множественных сред разработки — цель работы в стиле SOA.

Этот подход делает возможным создание систем, которые строятся вне служб. Службы представляют собой строительные блоки для приложения, которое может быть скомпоновано независимо от служебных действий. Как приложение, так и конечный пользователь или другая служба может использовать эти строительные блоки. SOA предоставляет возможность создавать диаграмму бизнес-процесса, в которой они обращаются к действиям службы.

Реализация приложения в этой архитектуре состоит в том, чтобы его код и функциональные возможности были пригодны для повторного использования в будущем. Так как бизнес-логика не привязана к какой-либо определенной технологии интерфейса пользователя, возможно обращение к функциям клиентов, которые пользуются более новыми технологиями для создания интерфейсов пользователя.

Есть и еще одно преимущество — разделение задач. При создании группы разработчиков для проекта различные подгруппы и индивидуальные члены могут быть назначены по обе стороны границ службы.

Одна команда сосредоточивается на разработке только средств взаимодействия с пользователем, не заботясь о коде, реализующем бизнес-логику и доступ к данным. Команда, отвечающая за пользовательский интерфейс (UI), получает интерфейс службы и может приступать к программированию этого интерфейса. Тем временем другая команда работает над реализацией службы без необходимости строить интерфейс пользователя. Это означает, что разработчик больше не отвечает за код целиком, включая интерфейс пользователя, бизнес-логику и доступ к данным в соответствии с заданными требованиями. В результате обеспечивается специализация разработчиков в какой-то одной области программного обеспечения.

Разделение задач также означает, что разработка пользовательского интерфейса (UI) и службы может быть начата одновременно, непосредственно после публикации согласованных интерфейсов служб. Это огромное преимущество, которое позволяет создавать пользовательский интерфейс (UI) за пределами компании, а внутри компании можно ограничиться созданием кода, относящегося к бизнес-логике.

SOA — это способ построения распределенных систем, в которых автономные операции вызываются с помощью слабосвязанных сообщений и четко определенного интерфейса.

Наличие строго определенного служебного интерфейса абсолютно необходимо. Преимущества SOA проявляются только тогда, когда договор о разработке службы согласован со многими сторонами и не изменяется в процессе разработки. Определение требований ко всем необходимым функциям — это уже бизнес. Это делается во взаимодействии с разработчиком функциональной архитектуры, который определяет интерфейс на техническом уровне. Конечно, это не всегда легко, а иногда даже невозможно, так как бизнес-требования весьма сильно изменяются в большинстве сред разработки. Чтобы разрешить это противоречие, разумно применить итеративный процесс разработки, который обычно длится от одной до четырех недель. Интерфейс службы не изменяется, а модификации интерфейса явно обсуждаются и сообщаются группам разработчиков при каждой новой итерации.

Так как приложения и системы программного обеспечения становятся больше и сложнее, нужна строгая архитектура развития, чтобы обеспечить бо́льшую ремонтопригодность с возможностью повторного использования компонентов. В наиболее распространенных современных средах разработки, где приложения реализованы на различных платформах, очень важно наличие простого подхода к развитию, который гарантирует взаимосвязанность.

Реализация архитектуры SOA нужна для разрешения проблем, которые сама лишь объектная ориентация не может решить в случае очень больших систем с интеграцией между различными частями. Интеграция существующих компонентов требует хорошо продуманной и всеобщей парадигмы в форме SOA.

Четыре принципа SOA

Чтобы лучше и детальнее определить SOA, некоторые принципы нужно описать более подробно. Принципы в индустрии программного обеспечения позволяют сделать это. В SOA обсуждаются четыре принципа ориентации на службы.

Эти принципы включают следующее.

- Границы задаются явно.
- Службы автономны.
- Службы имеют общую схему и соглашение, но не класс.
- Совместимость служб основана на стратегии.

Границы задаются явно

При работе по методологии SOA должны быть явно определены границы, которые должен пересечь клиент, чтобы добиться выполнения функций служб. Службы запускаются в пространстве процессов и памяти отдельно от клиентов, использующих их. Необходимо заранее определить границы и ознакомить с ними всех возможных участников. Границы определяются смыслом соглашения и адресом, по которому может быть найдена служба. Эта информация должна считаться важной и быть легко доступной.

Невозможно выполнять процедуры службы без наличия соглашения и адреса. Процедуры могут выполняться только одним способом, а именно путем вызова соглашения (контракта), которое рассматривается как граница. Явные границы означают, что клиент должен знать только о существовании функций в службе, которые могут быть

выполнены только через соглашение. Этот принцип также означает, что все возможные исключения должны быть описаны и метод может завершить выполнение, только если получен нужный ответ в виде четко определенной структуры данных или в виде структуры, содержащей детали исключения. Никакие данные не являются ни входными, ни выходными для службы без явного на то указания.

Службы автономны

Службы рассматриваются как автономные части кода, не зависящие от поведения других служб. Службы считаются общедоступными без необходимости явного создания их экземпляров. Следующие версии должны устанавливаться и разрабатываться независимо друг от друга, и установка новой версии службы не должна влиять на поведение других служб. Службы не нужно связывать друг с другом, как классы связываются в исполняемой программе, вместо этого они должны пользоваться слабосвязанной архитектурой.

Службы совместно используют схему и соглашение, но не класс

Схема определяет действия службы и сигнатуру в независимом от платформы формате: имена функций, типы параметров и тип возвращаемого значения. Соглашение описывает метаданные для службы в виде черного ящика, в котором хорошо описан только интерфейс. Схемы представляют собой описание структуры параметров. Этот принцип явно указывает, что сам класс (в виде кода или нотации на языке UML) не используется совместно службами и их клиентами.

Так как SOA предназначена для взаимодействия различных частей, реализованных на различных платформах, нет смысла передавать код другим частям приложения. Во многих случаях этот код бесполезен. Внутренняя организация (поведение кода) класса не интересна клиенту. Приложение (или другая служба), которое использует службу, заинтересовано только в результате действия службы. Клиенты отправляют сообщения той части действий схемы, которая удовлетворяет условиям соглашения, чтобы получить этот результат.

Клиенты должны взаимодействовать со службой и служба с клиентом средствами явно определенного общедоступного интерфейса, включающего описание исключений. Каждая версия службы должна иметь свою версию интерфейса. После того как начата разработка интерфейса, он не должен изменяться, так как изменения интерфейса были бы результатом изменения в поведении некоторых действий службы и должны приводить к новой версии интерфейса.

Совместимость служб основана на стратегии

Данный принцип означает, что условия обработки сообщения определяются службой. Стратегия применяется для согласования элементов в процессе коммуникации, таких, как, например, формат сообщения и требования безопасности. Стратегия способствует установлению семантики служб и ожидаемого поведения клиентской части.

“Анатомия” службы

Как и человеческое тело, служба представляет собой совокупность различных частей, связанных в одно целое. Каждая часть имеет свое собственное назначение и поведение в системе. Описание этих частей поможет вам понять, как работают службы, и станет введением в технические детали реализации SOA (рис. 1.1).

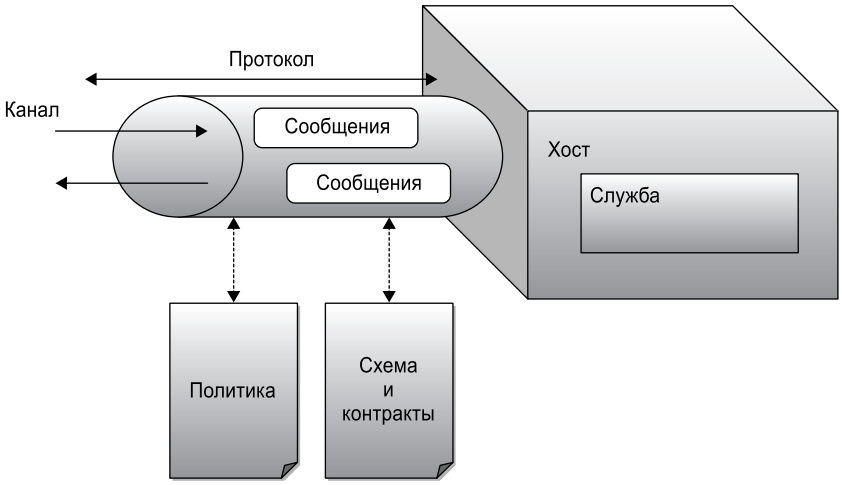


Рис. 1.1. Работа службы

Служба, в которой содержатся методы, пользуется каналом для того, чтобы его могли найти и использовать клиенты службы. Клиенты службы также пользуются каналом, который совместим с каналом службы, чтобы фактически вызывать методы и отправлять нужные данные службе. Канал представляет собой комбинацию схемы, соглашения и стратегии с одной стороны и используемого (во время выполнения) протокола с другой. Сообщения посылаются через этот канал в любом направлении. Канал привязан к протоколу определением, в каком направлении и как достижима служба. Протокол, например HTTP или MSMQ, в данном случае перемещает данные и поддерживается операционной системой, на которой реализованы службы. Канал — своего рода труба, по которой передаются сообщения. Сообщения посылаются в канал клиентами (или иными пользователями службы) и извлекаются из канала управляющим стеком платформы, которая публикует служба.

Канал привязан к схеме, с которой он согласован. Канал не является полным без описания метаданных действий службы, содержащихся в схемах и соглашениях.

Канал также знает стратегию, которую должен реализовать потребитель службы.

Экосистема службы

На верхнем уровне служба находится в экосистеме, в которой несколько понятий формируют часть парадигмы SOA. Эта экосистема описывает место этих понятий и их взаимодействие друг с другом (рис. 1.2).

Приложения формируются из служб

Сердце экосистемы — это сама служба. Службы — это строительные блоки, из которых формируются приложения. Приложения могут быть программными средствами для конечного пользователя с четкими частями пользовательского интерфейса или бизнес-процессами, которые обращаются к службе в определенной последовательности. Так как подход SOA позволяет рассматривать службы как единичные действия (составные единицы поведения), можно написать приложения, выбирая нужную службу, и тем самым компоновать другую часть программного обеспечения.

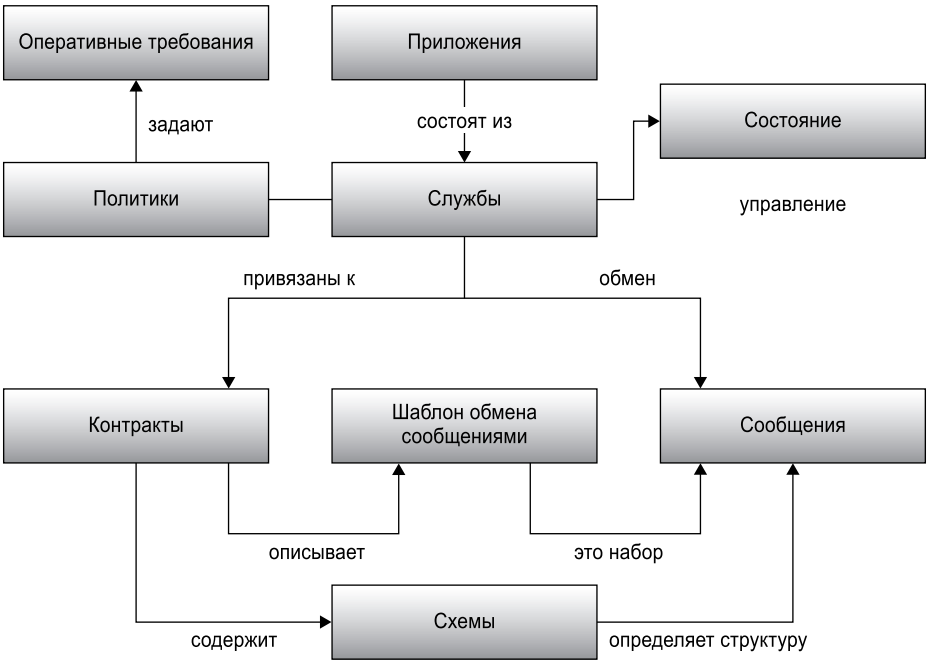


Рис. 1.2. Экосистема службы

Службы управляют состоянием

Часто службы и их операции отвечают за сохранность данных в базе и за чтение данных в последующем. Служба фактически управляет состоянием. Хотя службы не запоминают текущее состояние в том смысле, что они не хранят данные в памяти, они обращаются к базе данных, чтобы сохранить текущее состояние. Службы не обращаются к базе данных непосредственно, но в хорошей архитектуре они будут взаимодействовать с базой данных на других уровнях.

Службы задают политику

Служба имеет право определять политику использования ее логики. Политика описывает необходимые требования к поведению потребителя службы. Политику можно представить как соглашение об условии, которое должно быть выполнено перед тем, как клиенту будет разрешен доступ к службе. Чаще всего это договор о безопасности.

Политика предопределяет функциональные требования

Определяя политику, служба может предписать функциональные требования вызывающей платформе. Политика может требовать, чтобы на клиенте был реализован определенный уровень безопасности.

Службы ограничены соглашениями

Службы существуют только тогда, когда существует соглашение, описывающее сигнатуры его действий. Соглашение представляет собой договор (контракт) между клиентом и службой. Соглашение должно быть четко определено и связываться с службой во

время выполнения. Контракт необходим для создания класса модуля доступа на клиентском уровне, используемого при программировании службы так, как если бы это был локальный класс. Без соглашения (контракта) служба не может использоваться клиентскими приложениями.

Соглашения описывают протоколы обмена сообщениями

Протокол обмена сообщениями представляет собой определение того, как и в какой последовательности сообщения передаются от отправителя получателю. В зависимости от протокола служба может вызываться синхронно или асинхронно и ожидать результаты действия. Протокол обмена сообщениями может быть следующим.

- **Вопрос-ответ.** Наиболее часто используемый образец; в ответ на каждое сообщение должно быть получено другое сообщение.
- **Однонаправленный.** Не ожидается никакого ответа от службы, и поэтому действие может вызываться асинхронно.
- **Двунаправленный.** Действие службы может обращаться к клиенту в процессе вызова метода. Таким образом, действие службы может запросить дополнительную информацию у клиента перед возвращением окончательного ответа.

Протокол обмена сообщениями видим на функциональном уровне, на котором разработчики реализуют действия службы. На более глубоком техническом уровне протокол обмена сообщениями присутствует, но он по большей части невидимый. Пользуясь `wsHttpBinding` для запроса действия, клиент и служба не только обмениваются сообщениями с данными. Некоторые другие технические сообщения предшествуют вызову функции и завершают его. Мы можем рассматривать эти сообщения как сообщения подтверждения связи (рукопожатия), которые являются частью WS-протокола. Эти сообщения могут устанавливать контекст безопасности. Они также нужны для подтверждения того, что служба получила данные и может полагаться на то, что клиент получил ответ, как ожидается. Этот дополнительный обмен сообщениями не должен определяться разработчиком, его осуществляет протокол.

Соглашения содержат схемы, а схемы определяют структуру сообщения

Схема представляет собой определение структуры параметров для операции в виде файла, имеющего формат XSD. (XSD — язык метаданных, описывающий сообщения, поступающие к действиям службы, или результат действий службы.) Схемы применяются для того, чтобы клиенты операций служб могли форматировать данные. Эти данные могут интерпретироваться при вызове службой, так что он может обращаться к данным снова. Это называется сериализацией (преобразованием в последовательную форму) и десериализацией.

Протокол обмена сообщениями — это набор сообщений

Комбинация и последовательность вызова сообщений может быть описана более сложным протоколом обмена. Таким образом, протокол обмена сообщениями может определить, какое действие должно быть вызвано первым, а какое — последним, и можно ли определить последовательность выполнения операций полностью.

Службы обмениваются сообщениями

Обмен сообщениями — самая главная часть в экосистеме службы. Обмен сообщениями означает вызов операции и получение ответа (или генерацию исключения). Обмен

сообщениями — это способ вызова операций на другой машине. Одни сообщения передают входные параметры от клиента к службе, а другие передают ответы клиенту (в обратном направлении).

Координация служб в бизнес-процессах

Поскольку службы являются строительными блоками приложений в служебно-ориентированной архитектуре, действия службы также могут вызываться потоком работ (расписанием вызовов). Это расписание содержит определение последовательности и зависимостей между входящими и исходящими вызовами. Поток работ организует взаимодействие клиентов и служб для выполнения бизнес-процесса. Поток работ знает соглашение и схему служб, которые он использует или реализует.

Поток работ подобен программированию последовательностей и ветвлений в получающих операциях и вызову других действий в других службах. Вместо программирования таких последовательностей и ветвлений на языке программирования поток определяется декларативно на метаязыке, который может интерпретироваться, что во время выполнения позволяет интегрировать (собрать воедино) различные части приложения. Таким образом, расписание вызовов описывает весьма целесообразный и сложный протокол обмена сообщениями. Так как главная цель SOA — построение повторно используемых компонентов для приложения, использование расписания вызовов позволяет определить логику интеграции и бизнес-процессы.

Требование бизнес-процесса формирует основу определения соглашений и схем. Просмотр этих требований должен быть началом процесса анализа.

На рис. 1.3 приведены примеры согласования.

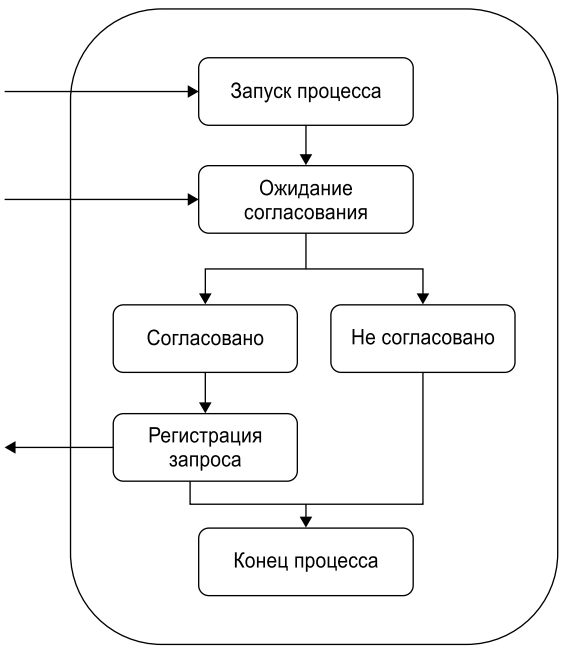


Рис. 1.3. Согласование

Технологические основы SOA

Чтобы разработать приложение в SOA, нужны технологии и протоколы. Поскольку SOA предназначена для создания распределенных и кроссплатформенных приложений, эти поддерживаемые технологии и протоколы должны быть промышленными стандартами. В следующих разделах описываются некоторые из самых популярных стандартов в мире SOA.

SOAP

Simple Object Access Protocol (SOAP) представляет собой спецификацию XML обмена данными в виде структурированной информации в сообщениях. SOAP стандартизирует способ обмена данными и не зависит от платформы, так как основан на XML. Сообщение SOAP просто переносит данные в виде сообщения. Конверт SOAP содержит (необязательный) заголовок и (обязательный) элемент — тело. Заголовок может содержать информацию, нужную для выделения технической инфраструктуры для поддержки коммуникации, и не связан с бизнес-функциональностью. Элемент (тело) содержит функциональные данные как полезную информацию. Каждый параметр для действия службы представлен в теле в виде последовательности данных. Пример сообщения SOAP приведен ниже.

```
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
    <m:StockName>XYZ</m:StockName>
    </m:GetStockPrice>
    </soap:Body>

    </soap:Envelope>
```

WS- протокол*

SOAP — всего лишь спецификация формата функциональных данных в теле и технических данных в заголовках. SOAP сам не всегда определяет значение заголовка. WS-* — это набор протоколов, которые стандартизируют реализацию определенных требований и режимов при работе с распространяемыми сообщениями, пользуясь сообщениями SOAP. Эти протоколы описывают, как канал должен выполнять обмен сообщениями безопасно, транзакционно и надежно, пользуясь заголовками в сообщениях SOAP. WS-* — набор протоколов, причем каждый протокол имеет свое предназначение.

WCF может реализовать WS-* протоколы с помощью соединения под названием `WsHttpBinding`. Это соединение использует часть WS-* протоколов и добавляет нужные свойства, например вызов транзакционных сообщений, надежность, обнаружение и адресацию.

WSDL

WSDL представляет собой описание соглашений в формате XML и содержит все метаданные для интерфейса службы, включая имена функций, имена и типы параметров, а также типы возвращаемых значений. Файл WSDL определяет соглашение независимо

от платформы, так как типы определяются в типах XML. WSDL может быть использован в средах разработки, отличных от .NET, для создания таких классов на их языках программирования (например, J2EE), которые будут служить заместителями действительно централизованного класса выполнения.

На рис. 1.4 показан пример файла WSDL. Этот файл может быть отображен браузером, если просматривать URL, на котором служба WCF отображает метаданные.

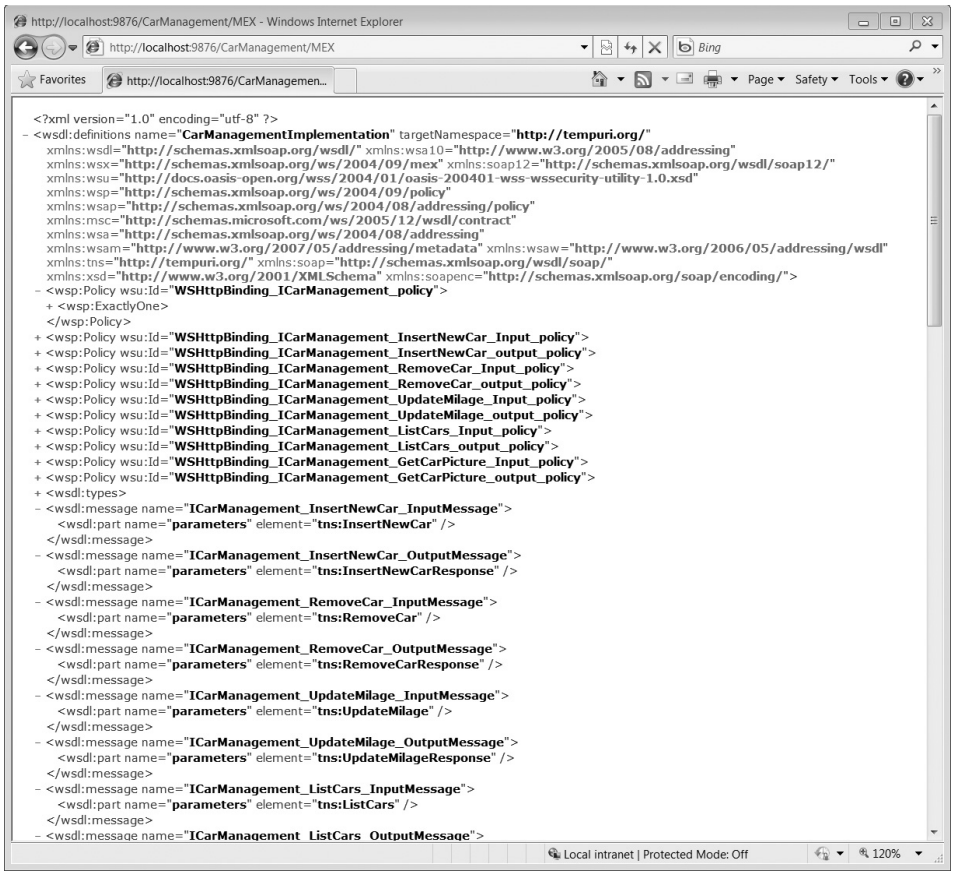


Рис. 1.4. Пример файла WSDL

Соглашение — это самое главное

Проектирование служб для приложения начинается с анализа требований. Не стоит сразу открывать Visual Studio и немедленно начинать программирование. Аналитики должны сначала поработать вместе с людьми, которым нужно приложение, и записать то, что они говорят, а затем сформулировать требования к службам. Встречи со всеми посредниками нужны для определения того, что должны делать службы и что представляет собой их логика.

Сначала нужно четко определить соглашение (контракт) службы. Анализ ориентированного на службы приложения состоит не в том, чтобы нарисовать экраны пользовательского интерфейса, определить таблицы и построить реляционную диаграмму. Он в первую очередь состоит в определении соглашения.

Конечно, в результате функционального анализа в приложении должны быть выделены три основных слоя.

- Пользовательский интерфейс (UI). Сюда относятся экраны, логика проверки правильности (валидности) и взаимодействие пользовательских средств управления.
- Логика. Здесь необходимо предусмотреть реализацию требований, бизнес-правила, вычисления и отчеты.
- База данных хранит данные и отвечает за целостность ссылок на данные в таблицах.

Ясно, что при подходе SOA в первую очередь должны анализироваться не пользовательский интерфейс (UI) и не база данных. SOA фокусируется на бизнес-аспекте и позволяет отделить логику от внешнего вида экранов и способа хранения данных. Ни пользовательский интерфейс (UI), ни база данных в действительности не связаны с бизнес-требованиями. В большинстве случаев бизнес не интересуется тем, как хранятся данные или как они будут представлены конечному пользователю.

Логика, реализованная в службах, должна быть доступна многим пользовательским интерфейсам. Интерфейс пользователя разрабатывает другая команда, которая ответственна только за пользовательский интерфейс (UI) и не имеет никакого отношения к бизнесу. Помните, что службы — это черные ящики; они используются для получения результата.

Логика не должна зависеть от способа хранения информации в базе данных. Администраторы DBA должны представить структуру таблиц так, как если бы база данных была уже готова.

Это приводит к первому принципу соглашения. Первое, что нужно проанализировать, — это соглашения. Вы должны иметь четкое представление о том, чего ожидать от службы, какие методы являются частью службы и что представляет собой структура параметров этих методов. Дизайн этого соглашения находится под сильным влиянием бизнеса и должен контролироваться им. Определение соглашений является задачей деловых аналитиков и аналитиков программного обеспечения.

Как WCF и службы .NET реализуют схемы в SOA

Схемы — это описание повторно используемого решения хорошо известных проблем в типичных ситуациях. Идея создания схемы целей состоит в описании решения проблемы на понятном языке перед тем, как оно будет переведено в код. Можно сказать, что схема целей — проект или шаблон для возможного решения. Она используется для обсуждения проблемы перед программированием на определенном языке и независима от инструмента разработки или использованной технологии.

Модели

Модели — это языки, использованные архитекторами или разработчиками нужных интеграций, которые люди могут обсуждать и использовать в их коммуникации.

Фразы типа “я предложил бы использовать расширенную модель перед отправлением сообщения брокеру...” и “для этого функционального требования я не вижу потребности в корреляционной модели здесь...” типичны в беседах, где обсуждаются служебно-ориентированные или основанные на интеграции решения.

Эти фразы достаточно понятны для проектировщиков процесса и архитекторов и вполне пригодны в ходе общения до запуска процесса разработки. После того как

проектировщики договорятся о плане, они могут нарисовать его в схеме. Модели проектов часто имеют дубликат в виде схемы с большой диаграммой, описывающей целую картину коммуникации и интеграции. Такая схема содержит комбинацию моделей и является идеальной основой для разработчиков в процессе создания кода и инженеров-системотехников в процессе реализации.

Разделенное соглашение: интерфейс и реализация

WCF поддерживает отделение и разработку интерфейсов, используя интерфейсы в качестве части языка C# или VB.NET. Все операции, типы их параметров и возвращаемых значений для службы могут быть выражены в интерфейсах, которые могут быть созданы в отдельном проекте библиотеки классов. Соглашения о данных, описывающие структуры параметров или возвращаемых значений, также описываются в классах.

Обычно эти интерфейсы и классы хранятся в библиотеке классов, которая не содержит никакой реализации. Реализация служб осуществляется в другой библиотеке классов. Это только вопрос ссылки на нужную библиотеку интерфейсов в проекте реализации и реализации интерфейсов в классическом виде, который поддерживают языки .NET.

Такое отделение позволяет использовать интерфейсы совместно с другими проектами, которые не могут обеспечить реализацию интерфейса так, как это делается в клиентских приложениях, хотя им эти интерфейсы необходимы.

Интерфейсам и описаниям их операций приписываются атрибуты метаданных, так что WCF может распознать их как соглашения, являющиеся частью экосистемы служб. Эти свойства размещаются в начале элементов кода и формируют дополнительный слой метаданных, которые понимает WCF во время выполнения, что придает коду определенное значение в терминах соглашений и схем. Поэтому метаданные для определения этих соглашений и схем, содержащиеся в свойствах, отделяются от интерфейса, и только реализация знает интерфейс. Это позволяет отделить соглашение от реализации. В метаданных имена операций и структуры XML сообщения SOAP могут быть определены без необходимости указывать их в реализации.

Эти свойства следующие: `[ServiceContract Attribute]`, `[OperationContract-Attribute]`, `[DataContractAttribute]` и `[DataMemberAttribute]`.

Proxy-модель

Proxy-модель используется WCF благодаря разрешению создавать и использовать классы в проекте, который пользуется службами. Данный класс ведет себя как реализация, но фактически реализует тот же общий интерфейс, что и реализация настоящей логики. Так как интерфейс вместе с дополнительными метаданными для сообщений SOAP, определенными в атрибутах, также доступен в клиентском проекте, этот proxy-класс может имитировать реализацию, реализуя интерфейс.

WCF предоставляет класс `ClientBase<T>`, который наследуют proxy-классы. Этот базовый класс содержит логику, нужную для установления взаимодействия со службой и преобразования вызовов операций. Клиентские приложения генерируют через прокси сообщения SOAP и отправляют их службе, используя определенный канал связи.

`ClientBase<T>` — родовой (универсальный) класс, который должен знать вид интерфейса. Данный класс имеет защищенное свойство, видимое только в классах, которые наследуют его. Это свойство является каналом и родовым типом, используемым прокси. Таким образом, канал имеет тот же список методов, что и фактическая реализация логики на серверной стороне.

Модели OperationContext

Модели OperationContext предназначены для того, чтобы отделить входные параметры функций от технической информации, которую должен обработать метод. WCF предоставляет класс System.ServiceModel.OperationContext, необходимый при выполнении методов реализации, поскольку он позволяет получить контекст выполнения и сообщения для текущего метода. Этот класс может обеспечить выполняющийся метод информацией о вызове, такой, как ID сессии, заголовки поступающих сообщений в конверте SOAP, информация, позволяющая идентифицировать того, кто вызвал службу, а также результат выполнения авторизации вызова. Контекст содержит много информации, которая не является функциональной и поэтому не должна быть частью параметров DataContract вызываемого метода. Работая в режиме двусторонней связи, OperationContext предоставляет канал, который используется для обратного вызова клиентской программы в течение выполнения вызова. WCF 3.5 также поддерживает дополнительный контекст под названием WebOperationContext, который предоставляет метод с дополнительной информацией о запросе в терминах свойств Http.

Соглашения о параллельности

WCF поддерживает соглашения о параллельности путем одновременной реализации множественных интерфейсов, а не просто разрешая службе иметь множественные конфигурированные конечные точки. Конфигурация каждой конечной точки имеет свой собственный адрес и ссылается на один из интерфейсов. В результате единственная реализация службы реализует набор действий, поступающих от множественных интерфейсов, в которых некоторые операции резервируются для одного специфического интерфейса в комбинации с операциями, которые определяются в множественных интерфейсах. Список операций, доступных интерфейсу, определяется конфигурацией конечной точки, так как эта конфигурация несет информацию о том интерфейсе, который она представляет.

Конфиденциальность данных

Безопасность и конфиденциальность данных реализована в WCF чрезвычайно хорошо. WCF поддерживает межплатформенную безопасность, используя стек протоколов WS-* для безопасности уровня сообщения, и может дополнительно пользоваться транспортной безопасностью на более высоком уровне в добавление к безопасности, достижимой через эти WS-* расширения. Уровень безопасности определяется выбранным соединением.

Атомарные служебные транзакции

WCF поддерживает транзакции, реализуя протокол WS-Atomic Transaction (WS-AT). Чаще всего транзакции на уровне службы ведут к транзакциям на уровне базы данных. WCF обменивается информацией с координатором распределенных транзакций базы данных, чтобы установить транзакции. Спецификации WS-Atomic Transaction определяют механизмы выполнения транзакций через границы службы, что позволяет транзакциям распространять вызов на клиента и распространяться дальше от фасадного метода сессии. Это позволяет клиенту запустить транзакцию, вызывать различные действия службы, причем все вызовы операций можно рассматривать как одну транзакцию. Транзакция распространяется от клиента к службе, и операция службы участвует в транзакции (иногда говорят “следует за транзакцией”), запущенной клиентом.

Таким образом, служба поддерживает транзакцию активной, пока клиент не решит, что она выполнена. Когда это происходит, служба посылает сигнал координатору распределенных транзакций, чтобы завершить транзакцию.

Распределение транзакций также означает, что если в течение одного из вызовов, сделанных клиентом, что-нибудь происходит неправильно на уровне службы, то транзакция считается прерванной и поэтому откатывается назад.

Поведение службы относительно транзакций может быть определено атрибутом `[TransactionFlow]` операции в интерфейсе и атрибутом `[OperationBehavior]` реализации.

Фасад сессии

Фасад сессии в WCF создают классы реализации, формирующие первый блок кода, которого достигает клиент, выполняя действие службы. Данные классы реализации отвечают за вызов бизнес-логики в других уровнях приложения. Такими уровнями могут быть уровень бизнес-логики, процесса выполнения и доступа к данным.

Сокрытие исключений

WCF автоматически скрывает все исключения, происходящие внутри или после фасада, и не показывает их детали вызывающим клиентам. Когда исключения происходят на стороне службы, клиент только информируется о факте, что что-то не так. По умолчанию подробная информация об ошибке, внутреннем исключении и трассировке стека для клиента не видима. Он видит только самые общие ошибки. Получение этого исключения заставляет канал клиента перейти в состояние ошибки.

Это сделано не только из соображений безопасности, не только чтобы не выдать информацию о трассировке стека хакеру, но и потому, что посылка деталей исключения .NET клиенту в архитектуре SOA была бы бесполезна, поскольку клиенты не всегда являются клиентами .NET. Поскольку SOA является кроссплатформенным подходом, возможно, что клиенты работают не в среде .NET, а в J2EE или другой среде, где получение исключения .NET было бы бесполезно и не могло бы быть интерпретировано должным образом.

WCF поддерживает более независимый от платформы способ передачи информации об исключениях при ошибках в SOAP. Транспортировка ошибки SOAP и сериализация/десериализация контента определяется стандартом SOAP и является кроссплатформенной и более SOA-ориентированной.

Использование ошибок SOAP в WCF становится возможным в результате определения соглашения о данных как любого другого `DataContract`, используемого интерфейсом. У этого соглашения о данных может быть настроенная для клиента структура, содержащая только данные об исключении, и того, что служба хочет предоставить и считает полезным для его клиентов и конечного пользователя. Операции, которые отправят это пользовательское соглашение об ошибке, должны знать это соглашение, что означает добавление к операции атрибута `Fault Contract`.

Методы, в которых исключение .NET имеет место, должны перехватывать это исключение и транслировать его в этот `DataContract`. Реакция на это исключение возлагается на клиента.

Коммуникация и модели интеграции

Коммуникация и интеграция — самые важные аспекты WCF. Это те области разработки программного обеспечения, где WCF действительно выступает во всем блеске и

где его значение трудно переоценить, — это именно то, для чего действительно создан WCF. Разработчики могут использовать богатый набор возможностей WCF, которые помогают создавать распределенные приложения, которые обмениваются данными и интегрируются друг с другом.

WCF — аббревиатура от Windows Communication Foundation, она означает, что ядро WCF действительно предназначено для коммуникации. Далее вы узнаете, зачем необходима интеграция и различные способы использования коммуникации. Затем будут описаны модели обмена сообщениями для достижения интеграции.

Коммуникация и интеграция необходимы потому, что сложные программы уровня предприятия обычно составляются из большого количества различных приложений. У каждого из них есть свои собственные функциональные возможности, построенные в различных архитектурных стилях, использующих различные технологии, причем реализованы они на различных операционных системах. Нередко сложные программы состоят из широкого набора приложений, либо сделанных по заказу, либо купленных как пакет, работающий на различных платформах и в различных местах, при этом каждое приложение выполняет только небольшую часть бизнес-требований.

Вся коммуникация, приводящая к обмену данными, основана на посылке сообщений. Эти данные являются либо набором параметров вопроса для получения информации, либо информацией, которая является ответом на такие вопросы. Сообщения форматируются и преобразуются в последовательный режим согласно отраслевому стандарту, которым является стандарт, основанный на XML SOAP. Целью форматирования сообщения в стандарте SOAP является функциональная совместимость. SOAP понимается многими технологиями, используемыми для общения и интеграции. SOAP и дополнительные стандарты WS поддерживают не только кроссплатформенное форматирование, но и дополнительные функции, такие как безопасность, надежность и транзакционное поведение.

Коммуникация и интеграция являются техническими частями приложения, которые позволяют функциям в распределенной среде взаимодействовать друг с другом и которые знают, как обмениваться нужными им данными. Из множества небольших приложений и служб, каждый из которых может работать в своей среде и выполняет одну функцию, формируются большие приложения.

Во многих случаях приложения являются распределенными по своей природе. Службы реализуются в привязке к базе данных, приложения работают на настольных компьютерах, а данные должны перемещаться между различными подразделениями компании. В процессе перемещения данные дополняются, агрегируются, или где-то принимается решение о том, куда должны быть доставлены данные. Чаще всего данные передаются динамично.

Стили интеграции

На протяжении всей истории архитектуры программного обеспечения было разработано множество стилей интеграции, причем каждый из них основывался на возможностях и требованиях своего времени — наличии технологий и инфраструктуры связи, а также стандартизации протоколов.

Мы различаем четыре стиля.

- Импорт и экспорт файлов.
- Общая база данных.
- Дистанционные вызовы процедур.
- Шина сообщений.

Импорт и экспорт файлов

Когда не было никакой коммуникационной инфраструктуры, необходимость интеграции не была осознана. Существовало не так много приложений для интеграции, и у всех сред разработки были закрытые архитектуры, — единственно возможной интеграцией был экспорт данных в файл из одного приложения и импорт файла, содержащего данные, в другое приложение. Файл мог быть с разделителями или позиционным сплошным файлом.

Первоначально это был ручной процесс, т.е. пользователи могли экспортировать и перемещать файлы на диск или использовать соединение FTP, позволяющее получающему приложению прочитать файл, а затем импортировать его вручную. Этот процесс позже был автоматизирован с помощью скриптов, запускаемых операционной системой или пользовательскими приложениями, которые могли выполнять макросы в приложениях и автоматически устанавливать соединения FTP.

Многие из этих приложений все еще используются во многих приложениях уровня предприятия, поддерживающих эту первую волну интеграции. Понятно, что такая интеграция является просто соединением равноправных узлов, причем процесс интеграции сильно зависит от приложений на обоих концах канала связи.

Данный стиль позволяет выполнять обмен данными в режиме задержек, причем импорт и экспорт планируются на ночное время. Это было приемлемо на заре вычислительной техники, но в наше время для современного набора приложений необходима быстрая интеграция в реальном времени.

Общедоступная база данных

Другой стиль интеграции был построен на основе общей базы данных. Вместо экспорта в файлы данные хранятся в формате централизованной базы данных, откуда они могут быть получены другими приложениями, имеющими доступ к этой же базе данных. Это стало возможным потому, что доступ к базам данных стал стандартизированным и большинство баз данных стали открытыми и доступными через связующее программное обеспечение (читай ODBC) на различных платформах. Интеграция в этом стиле позволила в большей степени сосредоточиться на многопользовательском аспекте применения, причем различные приложения получают доступ к этой общей базе данных одновременно, а данные передаются почти сразу, а не импортируются и экспортируются по расписанию.

Данный стиль также позволил приложениям использовать реляционную структуру базы данных. Изменение структуры данных в одной из частей может повлиять на правильность интеграции других частей. Для данного стиля интеграции важно наличие одинаковых структур в базе данных.

Дистанционный вызов процедур

По мере развития стилей стало ясно, что приложения должны взаимодействовать без использования процедур импорта и экспорта и что в больших сценариях интеграции общую базу данных обрабатывать тяжело. И тогда на смену пришел стиль дистанционного вызова процедур.

В стиле RPC (дистанционный вызов процедур) приложения могут передавать данные в другие приложения простым вызовом их методов, а также путем использования общих областей памяти и установления перекрывающихся физических границ сети. Для этого стиля интеграции были созданы новые технологии, например CORBA. Но

при использовании этого первого поколения технологий RPC возникла проблема: их реализация пригодна только для них и не доступна на других платформах.

SOAP впоследствии позволил найти решение не только в области SOA, но и доказал свою значимость в области интеграции. Но опять-таки RPC, даже с открытой технологией, имеет свои недостатки. Страх слишком тесной связи приложений через RPC-вызовы есть еще и сейчас. Еще один недостаток заключается в создании большого количества соединений между несколькими приложениями, где каждый новый результат интеграции ведет к созданию кода на стороне клиента и на другой стороне кода для получения данных.

Шина сообщений

Последний стиль — это шина службы (или шина сообщений). В этом стиле новая инфраструктура представляется как шина, по которой сообщения могут быть отправлены одним приложением и получены другим. Это может быть сделано путем публикации и подписки на получение моделей. Данные публикуются на шине, и другие приложения могут подписаться на получение данных по шине. Даже если нет доступных подписчиков, подписанных на данные, шина может взять на себя инициативу и отправить данные позже. С другой стороны, шина может также выступать в качестве читателя "внешних данных" и пересылать их подписчикам.

Публикация данных и подписка на них основаны на схеме, объявленной в соглашении о данных. Чаще всего файл XSD используется не только для объявления структуры сообщений, но и в качестве единицы подписки. Подписка на шину службы осуществляется путем указания на заинтересованность в определенном пространстве имен или определением имени корневого элемента в схеме XSD. Все данные, поступающие в шину в виде XML-сообщений, соответствуют этой схеме и автоматически передаются приемнику. Преимуществом шины является то, что новые приложения, заинтересованные в получении данных, уже опубликованных на шине для другого сценария интеграции, могут быть просто подключены к шине без необходимости изменения кода отправителя и переустановки клиентских приложений.

Основным преимуществом шины является то, что, если данные уже опубликованы на шине для другой программы, они могут быть использованы новым приложением, которое может подписаться на те же данные. Даже если данные, которые уже отправлены на шину, не вполне отвечают потребностям, потому что принимающее приложение требует не все или, возможно, дополнительные данные, такой подход позволяет продолжить работу. Данные могут быть отфильтрованы и объединены с другими данными, проходящими по шине, благодаря чему можно сформировать новый поток данных, который может использовать подписчик, подписавшийся на этот поток данных.

Шина службы позволяет создавать новые объединения приложений в будущем, не влияя на уже существующие интеграции.

ИНТЕГРАЦИЯ УСТАРЕВШИХ ПРИЛОЖЕНИЙ В НОВУЮ СРЕДУ

Устаревшие приложения — это программное обеспечение, написанное несколько лет назад с помощью технологий, существовавших в то время. Эти приложения не были предназначены для повторного проектирования, равным образом

их никто не собирался выбрасывать, когда появились новые требования, потому что инвестиции на переписывание их на основе новых технологий были бы слишком большими.

Такие приложения обычно закрыты и монолитны, зачастую сильно привязаны к набору данных или пользовательскому интерфейсу. Чаще всего эти типы приложений устойчивы, отлично выполняют свою работу и являются ядром программного обеспечения предприятия.

Когда необходимо удовлетворить новым требованиям, эти приложения остаются неизменными, но требуется интеграция с новыми приложениями или службами, созданными для удовлетворения новых требований.

Интеграция необходима для обмена данными — устаревшее приложение экспортирует данные, а новая служба нуждается в этих данных, чтобы удовлетворить требованиям бизнеса.

Модели обмена сообщениями

Модели обмена сообщениями (иногда называемые MEP) являются описанием общих способов достижения двух целей: коммуникации и обмена сообщениями.

Модель "вопрос–ответ"

Это наиболее часто используемая модель. Клиент запрашивает информацию у сервера, отправив сообщение с запросом, и ожидает ответного сообщения от службы. Это означает, что служба должна быть доступна и клиент ждет ответа в течение определенного времени. Клиент четко знает структуру ожидаемого ответа на вопрос. В случае передачи исключения во время обработки запроса, будет отправлено сообщение об ошибке (рис. 1.5).

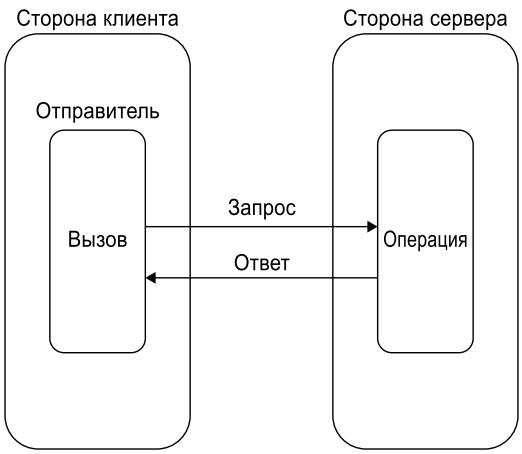


Рис. 1.5. Модель "вопрос–ответ"

В большинстве случаев модель "вопрос–ответ" используется для организации связи между приложениями. Почти все коммуникации клиента представляют собой запрос данных от службы, который отправляет ответ.

WCF поддерживает эту схему, позволяя разработчику определить метод в C# или VB.NET, причем методы имеют один или более входных параметров и могут иметь тип возвращаемого значения. Эти параметры могут быть любого типа, такого, как примитивные строки и целые числа, и более сложными типами, определенными в классах.

Подход WCF состоит в том, что спецификации этих методов могут быть определены в интерфейсе, являющемся элементом языка C# или VB.NET. Этот интерфейс содержит спецификации методов вместе с атрибутами, чтобы показать, что метод доступен для дистанционного клиента. Поскольку имеется управляющий класс, реализующий этот интерфейс, WCF поддерживает разделение метаданных, указывая, что они являются операциями в практической реализации. Классы реализуют только интерфейс и не знают, что они вызываются дистанционными клиентами. Такими атрибутами являются `[ServiceContract]` для интерфейса и `[OperationContract]` для спецификации методов.

Атрибуты интерфейса указывают, что методы реализуют модель "вопрос–ответ". Атрибуты имеют дополнительные параметры для дальнейшего определения подробностей запроса и ответа, например пространство имен для соглашения и значения SOAP-действий и ответных действий.

Однонаправленная модель

Сообщения передаются от клиента к серверу в одном направлении. Клиент отправляет сообщение, но не ожидает ответа. Во многих случаях он даже забывает об отправке сообщения. Сервер просто обрабатывает сообщение, не отвечает на запрос и не отправляет подтверждение обратно клиенту. Понятно, что не все коммуникации можно сделать в этой модели. Тогда, когда необходим прямой ответ на запрос, эта модель не пригодна к использованию (рис. 1.6).

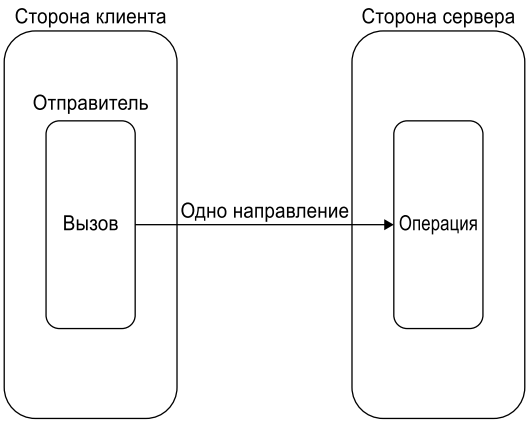


Рис. 1.6. Однонаправленная модель

Самым большим преимуществом этой модели является возможность работы в асинхронном режиме, что позволяет клиенту продолжать свою работу в то время, когда получатель обрабатывает сообщение в собственном темпе. Создается впечатление, что взаимодействие происходит очень быстро, так как клиент реагирует сразу же после отправки запроса, но обработка запроса возлагается на службу. Обработка на стороне сервера может выполняться в многопоточном режиме, чтобы служба могла обрабатывать несколько сообщений одновременно.

Еще одно, пожалуй, иногда даже еще более важное преимущество состоит в возможности создания изолированной среды доставки. При использовании односторонней модели можно отправить сообщение через системы управления очередями, такие как MSMQ. Очередь на клиентской стороне получает сообщение и пытается отправить его получателю. Когда получатель не доступен по той или иной причине, этот механизм дублирует сообщение и доставляет его, когда другая сторона снова станет доступной. Система управления очередями может быть сконфигурирована таким образом, что будет сохранять данные, когда доставка невозможна, — сообщения сохранятся после перезагрузки.

То, что метод выполняется в однонаправленной модели, определяется частично на уровне самой операции, указанием типа возвращаемого значения как `void` или определения метода в качестве подпрограммы вместо функции в VB.NET.

Это должно быть сделано в сочетании с указанием на использование односторонней модели в интерфейсе путем установления параметра `IsOneWay` равным `True` на атрибутах `OperationContract`.

Дуплексный обмен сообщениями

В дуплексной модели клиент и сервер обмениваются сообщениями без предопределенного шаблона. Хотя клиент и посылает начальный запрос к службе и дожидается ответа, он позволяет службе, которая отправляет ответ на запрос клиенту, запросить дополнительную информацию. Служба может направить обратный вызов клиенту динамически и, возможно, несколько раз, прежде чем отправить ответ на первоначальный запрос (рис. 1.7).

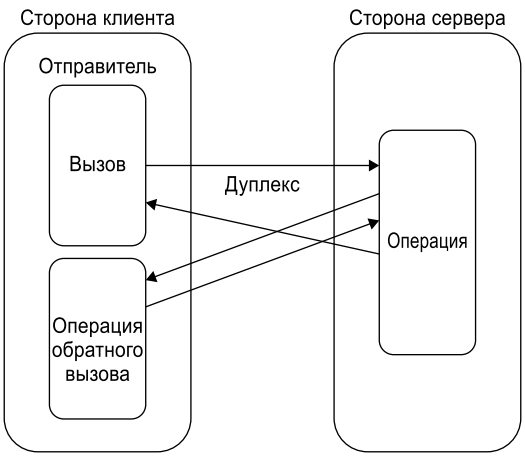


Рис. 1.7. Дуплексный обмен сообщениями

В этой модели клиент становится службой для кода во время исполнения. Код, который рассматривается в качестве службы, может теперь выступать в качестве клиента и вызывать функции во время обработки вызова.

Когда в дуплексной модели происходит обратный вызов клиенту во время выполнения, используется отдельный механизм связи. Кроме того, необходимо, чтобы сигнатура операции, которая вызывается службой, была доступна как службе, так и клиенту. Вначале служба должна использовать дуплексный способ сообщения, поскольку другого пути для обратного вызова клиента нет.

Это делается путем выбора одного из дуплексных каналов, поддерживаемых WCF, с указанием этого в конфигурации. Сигнатуры методов, которые реализуются в клиенте, должны быть определены на уровне службы и реализованы на клиенте.

WCF позволяет определить соглашение обратного вызова как обычный интерфейс и указывает в интерфейсе службы, что он может вызвать один из своих методов в интерфейсе обратного вызова. Это делается путем указания атрибута `[Service Contract]` в службе типа интерфейса обратного вызова.

Таким образом, определение соглашения обратного вызова является частью файла WSDL. Клиент может реализовать класс на базе этого интерфейса. Служба точно знает, какие методы он может использовать для обратного вызова клиента, поскольку ему известны сигнатуры методов, доступных на стороне клиента.

Дуплекс означает, что служба временно работает в качестве клиента и должен иметь канал доступа к нему. Во время выполнения метода служба имеет доступ к контексту, в котором содержится ссылка на этот канал обратного вызова. Вызов методов на этом канале инициирует вызов методов на стороне клиента.

Потоковая модель

В потоковой модели клиент инициирует запрос для получения очень большого набора данных. Служба разделяет данные на более мелкие части и отправляет их клиенту по одному. Данных так много, что службе приходится читать их из файловой системы или из базы данных частями. Части данных отправляются клиенту в отсортированном виде, в том, в каком получатель должен их обработать. Так должно быть в случае передачи потокового видео.

В этой схеме за одним запросом на получение данных следует большой набор ответов, каждый из которых содержит подмножество всех данных, являющихся результатом вызова. Только отправитель должен указать в последнем сообщении, что поток данных исчерпан, чтобы клиент не ожидал данных далее (рис. 1.8).

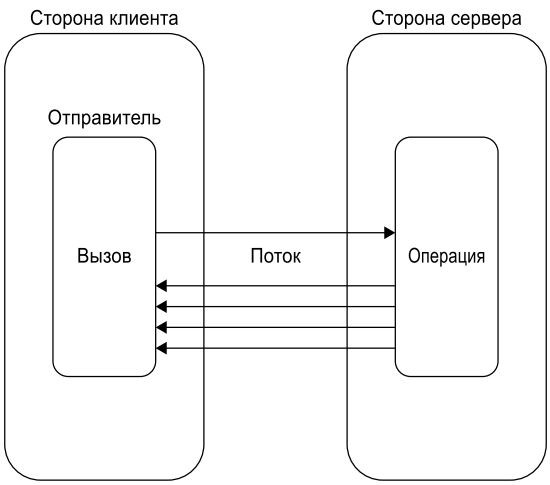


Рис. 1.8. Потоковая модель

Модель быстрых очередей сообщений (Pub – Sub)

В модели Pub – Sub вы видите приложения, публикующие данные обычным способом, а также ряд других приложений, заинтересованных в этих данных и

подписавшихся на эту публикацию. Например, система котировок акций, которая часто публикует котировки, получаемые заинтересованными сторонами.

В модели pub – sub тот, кто публикует данные, как правило, не знает, кто является подписчиком и что он делает с полученными данными. Издателю даже безразлично, где находятся подписчики, — он просто заботится о передаче данных из своей системы. Модель применяет событийно-ориентированный подход; издатель вызывает событие в клиентских приложениях. Эти клиентские приложения выполняют определенные действия в соответствии с этим событием, обрабатывая полученные данные. Эта обработка может быть различной: они могут отображать информацию конечному пользователю, могут хранить ее в своих хранилищах данных или же принять решение о направлении данных в другие приложения (рис. 1.9).

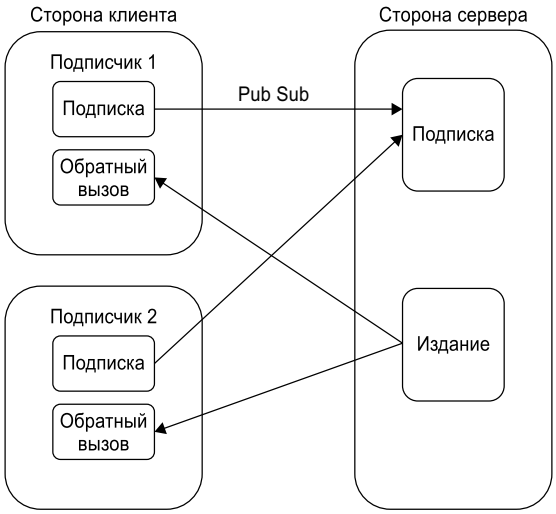


Рис. 1.9. Модель быстрых очередей сообщений

В модели pub — sub всегда есть явные действия, называемые подпиской, когда заявка выражает заинтересованность приложения в информации от издателя. Чаще всего подписчик не заинтересован во всей информации, которую посылает издатель. Подписавшись, подписчик может в качестве фильтра указать условия, которым должны удовлетворять данные. Издатель сохраняет список ссылок на подписчиков с фильтрами, и каждый раз, когда у него есть новая информация, которая должна быть послана, он просматривает всех заинтересованных подписчиков и публикует данные. Во многих случаях издатель информации непосредственно не управляет подпиской и передачей данных. Вместо этого он посылает свои данные универсальный службе, которая отвечает за публикацию данных. Эта дополнительный служба управляет подпиской и может публиковать полученные данные всем подписчикам.

У WCF 4.0 нет никакой прямой поддержки модели pub – sub. Модель pub – sub может быть реализована при использовании той же самой конструкции, что и для двустороннего шаблона. Данная модель может быть реализована при наличии службы подписки, которая поддерживает в памяти список ссылок на каналы обратного вызова подписчиков.

Когда служба должна отправить данные всем подписчикам, он может выполнить просмотр всех подписчиков и вызвать метод передачи этих данных. В данном сценарии

нет никакого последовательного опроса, поскольку подписчики действуют как главный компьютер, ожидающий вызова методов от служб.

Службы .NET (в рамках AppFabric Azure) поддерживают модель pub – sub без использования дуплексной связи, а с помощью коммуникационного механизма, реализованного в соединениях eventRelay.

Клиенты могут регистрировать свой собственный адрес на шине службы .NET в облаке. AppFabric записывает, где находится клиент. Когда данные должны быть переданы всем подписчикам, на шину службы достаточно послать сообщение. Шина службы передает это сообщение всем подписчикам, поскольку знает все их адреса.

Вызов подразумеваемых последовательностей

Если для выполнения одной логической единицы работы в службе нужно выполнить несколько операций соглашения в некоторой последовательности, полезно определить последовательность вызова операций. Определяя подразумеваемую последовательность вызовов, клиент знает о ней, и это приводит к тому, что некоторые операции не могут быть выполнены прежде других.

Такое может происходить тогда, когда операции полагаются на состояние, созданное операциями, вызванными ранее. Подразумеваемая последовательность операций позволяет описывать, какие операции могут использоваться для запуска логической группы вызовов и какой метод может нарушить эту логическую последовательность в случае его вызова.

Эта модель также позволяет указать методы, которые считаются последними в логической последовательности. Запрос этих завершающих методов указывает клиенту, что бесполезно вызывать другие методы, поскольку логический модуль работы считается завершенным.

Данная модель не является моделью простого обмена сообщениями, поскольку у вызовов методов, участвующих в упорядоченных вызовах, может быть собственная реализация другой модели. Это скорее модель, группирующая множество операций в один логический модуль работы (рис. 1.10).

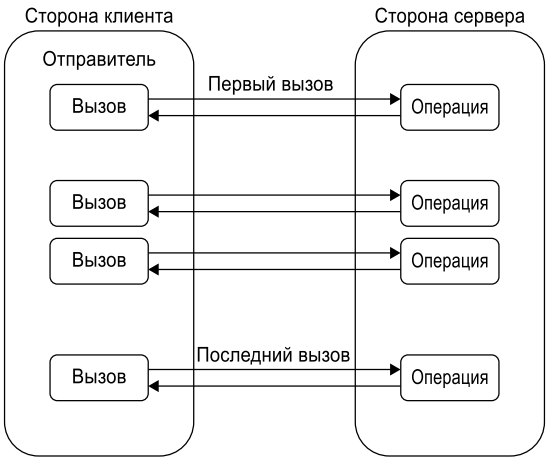


Рис. 1.10. Вызов подразумеваемых последовательностей

WCF предоставляет атрибут `OperationContract`, который позволяет определить ограничения на порядок вызова операций. Эта модель позволяет определить метод,

который может использоваться для запуска сеанса связи. Для этого значение параметра `IsInitiating` необходимо установить равным `false`. Задание значения `IsTerminating` равным `true` означает, что вызов данного метода указывает на окончание сеанса связи.

По умолчанию, если параметры не указаны явно, у каждого метода параметр `IsInitiating` установлен в значение `true`, а параметр `IsTerminating` — в `false`. Таким образом, каждый метод может быть вызван первым, причем ни один метод не заканчивает сеанс связи. Вы сами можете решить иначе и определить, какие методы не могут использоваться первыми (`IsInitiating = false`) и какие методы должны закончить сеанс связи (`IsTerminating = false`).

Здесь важно то, что клиент также знает эту последовательность вызовов, так как это определено на уровне интерфейса. Таким образом, прокси-сервер не будет вызывать методы, которые нельзя вызывать, либо потому, что первым был вызов метода, который не является начальным, либо потому, что вызов метода произошел после того, как уже был вызван завершающий метод.

Еще один способ объявить порядок вызова методов в службе — реализация его в виде службы *потока работ* (workflow). У него есть более гибкие модели задания порядка операций. Служба потока работ представляет собой декларативное задание последовательности операций в виде блок-схемы.

В таком определении можно явно указать последовательность методов, а также ожидаемое количество методов вне интерфейса, которые вызываются на определенном шаге процесса, — такие методы можно размещать в процессе прослушивания (класс `ListenActivity`). Процесс прослушивания (класс `ListenActivity`) группирует операции и останавливает процесс, пока не будет вызван один из методов. Этот вызов выполняет действия непрерывно с остальной частью процесса, в том смысле, что все другие методы в том же самом процессе прослушивания (класс `ListenActivity`) больше не доступны.

Такое определение порядка неизвестно клиенту. Это было бы слишком сложно и бесполезно для прокси-сервера, поскольку служба последовательности работ может вести себя очень динамически. Он может перейти к другому набору методов в процессе прослушивания (класс `ListenActivity`) на основании анализа входных данных, полученных от предыдущего запроса. Когда некоторый клиент вызывает метод, который в настоящее время не доступен, он получает сообщение об ошибках, указывающее, что требуемая операция не найдена.

В службах последовательности работ операции могут быть определены как инициаторы нового процесса или как участники существующего и уже выполняющегося процесса. Как правило, первая операция в последовательности работ инициализирует процесс, а другие методы следуют за ней.

Модели обмена сообщениями

Модели обмена сообщениями могут использоваться для различных целей, от наполнения контента до маршрутизации сообщений для перехвата фильтров, и выполнения других действий. Следствием этой многофункциональности и является эффективность архитектуры WCF во время выполнения. В следующих разделах модели обмена сообщениями рассматриваются более подробно.

Архитектура WCF во время выполнения

Чтобы подробнее изучить модели обмена сообщениями, вначале нужно знать, что архитектура WCF во время выполнения является многоуровневой, причем сообщения

передаются из приложения-клиента в поток, транспортирующий сообщение. Далее сообщение транспортируется до реализации службы на другой стороне.

В процессе перемещения сообщение проходит через прокси-слой клиента и слой диспетчера службы. Эта многослойная архитектура во время выполнения позволяет реализовать различные модели обмена сообщениями, что в свою очередь означает бо́льшую расширяемость для подключения пользовательского кода, в результате чего можно добавить логику или изменить поведение во время вызова операции. Другими словами, прокси-сервер транслирует вызовы методов в сообщения WCF, посылает их через канал, а диспетчер принимает сообщения WCF из канала и транслирует их в вызовы кода (рис. 1.11).

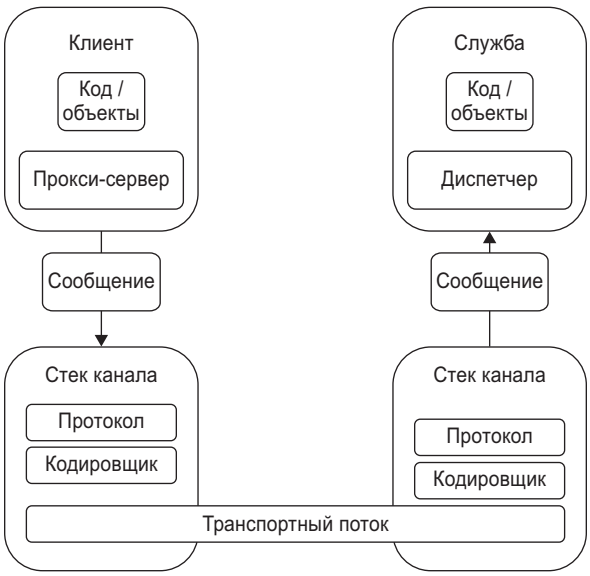


Рис. 1.11. Архитектура WCF во время выполнения

В этой архитектуре прокси-сервер направляет сообщения в канал. Находящееся в канале сообщение проходит другие уровни, такие как уровень протокола и уровень кодирования, перед тем, как оно фактически будет передано в поток для службы. На стороне службы сообщение будет получено соответствующим стеком канала, который посылает это сообщение через соответствующие уровни далее. Он проходит уровень декодирования и протокола и принимается диспетчером, который проверяет сообщение и определяет, какие методы вызывать.

Модель дополнения контента

Модель дополнения контента является шаблоном, который работает на функциональном уровне. Если приложения отправляют данные, содержащие не всю информацию, которую ожидает получатель, необходимо дополнение контента.

Причина этой нехватки информации может быть исторической — возможно, во время создания сообщения не было потребности включать эту информацию в сообщение, поскольку она была всегда доступна в базе данных. Но в распределенной среде не у всех служб есть доступ к базе данных. Между отправителем и получателем другая служба должна принять сообщение, исследовать его и добавить к нему необходимую

информацию. Поиск этой дополнительной информации может быть осуществлен средствами запроса к базе данных, путем выполнения бизнес-правила или просто жестким кодированием.

Дополнение данных является некоторой процедурой, которую выполняет реализация операции службы. Дополнение контента потенциально может означать, что соглашение о данных изменяется динамически. Поведение, направленное на дополнение данных, может привести к добавлению нового свойства к соглашению о данных, приданию этому новому свойству некоторого смысла, а затем посылки измененного соглашения о данных далее другим службам.

Реализация этого в универсальном подходе означает, что в WCF необходим способ получения сообщений, для которых не определено соглашение о данных в сигнатуре операции, и требуется посылать сообщение далее также при неизвестном коду соглашении о данных. Очень часто дополнение контента является просто универсальными функциями, получающими данные в универсальном формате (XML), изменяющими этот формат и посылающими данные далее другой службе.

WCF поддерживает получающие данные без контроля типа классом сообщения. Функция получает один параметр типа сообщения и возвращает сообщение или ничего в случае однонаправленного *пустого* (void) метода. Класс сообщения — своего рода каталог соглашения о данных. В операции с классом сообщения в качестве входного параметра `inputparameter` может быть получено любое сообщение, которое еще не указывает на другую операцию. Так, при создании службы, у которой есть одна операция с параметром-сообщением, выполняется дополнение контента.

Конечно, у этого кода нет прямого доступа к соглашению о данных, поскольку он не знает подробности о соглашении, но может читать данные на уровне XML, и изменить контент или добавить узлы XML где необходимо.

Маршрутизация сообщений

Во многих случаях отправитель сообщения не знает, куда послать сообщение. Фактический получатель не известен клиенту, и поэтому именно механизм маршрутизации определяет действительный адрес конечной точки. Маршрутизатор может выполнить различные виды обработки во время маршрутизации.

Это решение может зависеть от содержимого сообщения в сочетании с набором бизнес-правил. Или же фактическая конечная точка может только быть жестко закодирована. Во всех случаях идея состоит в том, чтобы разделить отправителей и получателей через организацию гибкого пути.

Когда фактическая конечная точка определена значением или комбинацией нескольких значений в содержимом сообщения, маршрутизация называется основанной *на содержимом* (content-based routing).

Часто бизнес-требование предполагает, что данные должны быть посланы одному из возможных множественных получателей, который интересуется этими данными и может выполнить запрос, в то время как другие получатели этого не могут.

С выходом .NET 4.0 в WCF появились хорошие механизмы поддержки маршрутизации сообщения. Такая поддержка состоит в том, что хосту службы разрешено использовать класс, который не имеет никаких функций, а только направляет сообщения другой службе, реализация которой находится в другом месте.

Реализация интерфейсов этого класса может находиться в пространстве имен `System.ServiceMode.Routing`, чтобы можно было определить необходимые шаблоны обмена сообщениями. Эта служба маршрутизатора определяет целевую службу, оценивая входное сообщение согласно набору фильтров.

Данная служба маршрутизации устанавливается и конфигурируется, как и любая другая. Это означает, что у нее есть множество конечных точек, определяющих адрес: соединение и соглашение. Вы можете конфигурировать службу, добавляя функциональность, которая управляет таблицей маршрутизации. Таблица маршрутизации представляет собой список ссылок на клиентские конечные точки в той же самой конфигурации вместе со ссылкой на фильтры. Таким образом, конфигурация точных условий маршрутизации выполняется на основе критериев фильтров, которые могут работать со значениями WS-адресов, чтобы определить маршрут, могут работать с именами конечных точек или использовать выражение XPath для оценки входных сообщений.

Механизм маршрутизации также поддерживает соединение протокола, определяя необходимые соединения в конфигурации целевых оконечных точек, на которые ссылается таблица маршрутизации.

Соединения протоколов

Когда сообщения, поступающие в маршрутизатор, форматируются в протоколе, который не может быть понят возможными получателями, на маршрутизатор возлагается ответственность за переключение на необходимый протокол.

Это делается путем чтения данных из сообщения в исходном протоколе и копирования данных в сообщение протокола адресата. Соединение может быть установлено в результате переформатирования данных из закрытого частного формата в XML согласно схеме. Но и на транспортном уровне соединение может переключаться от одного транспортного протокола к другому.

Функциональные возможности маршрутизатора WCF 4.0 могут также действовать как мост между протоколами. Использование определенного протокола определено соединением, сконфигурированным в конечной точке вместе с ее адресом. Чтобы изменить протокол и послать сообщение с запросом к другой конечной точке по другому соединению, может использоваться система маршрутизации WCF, которая автоматически транслирует протокол.

Фильтры сообщений

Фильтры сообщений являются ключом к конфигурированию маршрутизации сообщения. Они позволяют декларативным способом описывать условия, необходимые механизму маршрутизации, который принимает решения о том, куда направить входящие сообщения. С помощью фильтра сообщений можете объявить, что для некоторого действия в SOAP WS-адресации сообщение должно быть направлено другим службам. Поэтому можно иметь выделенную службу для каждого действия SOAP. Фильтры сообщений могут полагаться не только на действия SOAP, но и на другую информацию, необходимую для запроса. Запросы XPath, просматривающие содержание сообщения или заголовки SOAP, также полезны в качестве фильтров сообщений.

Резервная конечная точка

Маршрутизация может также быть полезной для реализации резервной конечной точки. В этом случае есть таблица маршрутизации, сконфигурированная для нескольких конечных точек, которые не связаны ни с каким фильтром сообщений и не используются для соединения протоколов.

Но в таблице маршрутизации определена альтернативная конечная точка, если первая конечная точка не доступна. Таким образом, обеспечивается запасной выход, если целевой служба, проходя по механизму маршрутизации, решает посылать сообщение в альтернативный маршрут.

Групповая передача

В такой модели механизм маршрутизации доставляет сообщение нескольким получателям. Для того чтобы это сделать, на сообщение накладываются фильтры. Входящее сообщение сравнивается со всеми фильтрами и посылается в конечную точку для каждого подходящего фильтра. Ясно, что групповая передача применима только для однонаправленного MEP, так как маршрутизация запроса для более чем одного получателя может привести ко множественным ответам клиенту.

Обнаружение служб

Обнаружение используется тогда, когда во время выполнения местоположение служб динамически изменяется, поскольку одни службы присоединяются к сети, а другие исчезают. Обнаружение может использоваться также тогда, когда размещение службы не находится под управлением клиентов и администраторов их сети.

В процессе обнаружения клиенты могут искать необходимый адрес самостоятельно, посылая в конечные точки службы в сети тестовое сообщение, которое содержит необходимые критерии для обнаружения соответствия. Обнаружение соответствия приводит к обнаружению службы и его сетевого местоположения в форме URI. Чаще всего этими критериями являются соглашения. Это означает, что содержимое WSDL службы используется при поиске службы, которая может соответствовать необходимому контракту.

Когда адрес конечной точки найден, клиент может обмениваться информацией с этой службой, так как служба имеет корректное соглашение.

Помимо этого узкоспециализированного способа работы, когда последовательный опрос транслируется по сети для нахождения подходящей конечной точки, в сети для службы есть более дружественный способ объявить о себе возможным подписчикам.

WCF реализует основанный на SOAP протокол обнаружения WS-Discovery и может сделать службу поддающейся обнаружению путем конфигурирования поведения служб. Это позволяет обнаружить службу в локальной подсети по UDP. Клиенты могут использовать протокол обнаружения WS-Discovery для поиска реального адреса выполняющейся службы.

Клиенты могут сделать это, используя класс `DiscoveryClient`. Данный класс предоставляет метод спецификации критериев поиска службы в сети. Чаще всего это будет тип интерфейса, который клиент хочет использовать для взаимодействия со службой. После того как соответствующая служба найдена, клиент получает доступ к адресу и может использовать его, чтобы определить адрес реального прокси-сервера, который используется для сеанса связи. Клиенты могут сузить результаты, определяя более детально диапазон поиска. Службы также могут объявить, являются ли они активными и присоединились ли к сети. Этот сигнал могут получить клиенты, которые реализуют протокол начала обнаружения WS-Discovery. Чтобы сделать это, клиент должен создать экземпляр `AnnouncementService`, у которого есть события, которые обнаруживают, когда службы становятся доступными в сети или исчезают. WCF также обеспечивает прокси для обнаружения, что позволяет построить центральную службу как хаб между множеством служб и клиентов.

Перехватывающие фильтры

Использование модели перехватывающих фильтров позволяет добавлять логику, которая выполняется сразу после получения вызова и непосредственно перед выполнением операции в службе. Это также может быть сделано непосредственно по окончании выполнения и перед отправлением ответа обратно клиенту.

Эта логика перехватывает вызов и делает некоторую предварительную обработку и постобработку. В этой обработке можно проверить аутентификацию, принять решение о создании сеанса, декодировать данные, при необходимости начать транзакции и предоставить значения по умолчанию для недостающих данных, если клиенты посылали данные в соответствии с устаревшими версиями соглашения о данных.

Цель состоит в том, чтобы централизовать и многократно использовать эту логику обработки для множественных вызовов операций в службе и не писать технический код для этой логики, смешанный с функциональным кодом реализации операции службы. Благодаря использованию перехватывающих фильтров, которые не выполняются на стороне сервера, ни клиент, ни службы не привязаны к этой технической логике и ничего не знают о выполнении обработки.

WCF поддерживает перехватывающие фильтры, реализованные в пространстве имен `System.ServiceModel.Dispatcher`, которое содержит классы и интерфейсы, позволяющие изменить выполнение приложения WCF на уровне диспетчера. Диспетчер получает сообщение от стека канала и вызывает эти методы. Это выполнение также может быть изменено пользовательским кодом.

Этот код может перехватить вызов и выполнить нужное действие перед реализацией, если операция уже выполняется. Есть дополнительные точки, позволяющие выполнить проверки параметров, преобразование сообщений, установить кэширование вывода, авторизацию, выполнить регистрацию сообщений и многих других шаблонов.

Создание точки расширения чаще всего сводится к созданию класса, который реализует один или несколько интерфейсов, найденных в пространстве имен диспетчера, и написанию пользовательского кода, реализующего интерфейсы.

Контекстные объекты

Модель контекстных объектов описывает, что во время выполнения вызова есть данные, доступные с техническими подробностями вызова, и экземпляр класса службы. Это означает, что реализация может не только положиться на входящие параметры функций, но что у нее также есть доступ к объекту, названному контекстом, где хранятся дополнительные технические данные.

Это данные о вызове вообще или о работающем экземпляре класса службы. Обращаясь к контексту объекта, операция получает доступ к информации о связанном канале, указывающем на клиента, возможном уровне защиты информации для пользователя, вызывающего операцию, информацию из заголовка сообщения, идентификатор сеанса и т.д.

Модели бизнес-процесса

В результате попыток описания бизнес-процесса в письменной или схематизированной форме были обнаружены некоторые повторяющиеся сценарии, которые можно считать общими шаблонами, применимыми в большом количестве сценариев и присутствующими почти во всех бизнес-процессах.

Менеджер процессов

Это модель бизнес-процессов высокого уровня. Она описывает инфраструктуру и понятия, необходимые для выполнения определенных шагов в бизнес-процессах, позволяет приложениям участвовать в процессе, обмениваться информацией с ним и достичь его цели.

Ясно, что процессами нужно управлять. Следовательно, есть среда, которая выполняет процессы, запускает экземпляры класса процесса, обрабатывает исключения, сохраняет состояние процесса и может перевести его в неактивный режим.

Процессы — это объявление списка шагов, которые будут предприняты для достижения цели. Цель собирает информацию, принимает решения на ее основе и распространяет ее.

Запуск процесса вызывается некоторым событием в бизнесе. Например, процесс должен быть начат, когда новый служащий начинает работать в компании. Цель процесса состоит во множественной регистрации в различных приложениях, в назначении нового служащего менеджеру, предоставлении новому служащему доступа к intranet-приложениям и разрешению ему заказывать автомобиль.

Многие действия происходят тогда, когда служащий начинает работать. В некоторых случаях действия должны произойти в предопределенной последовательности, когда невозможно перейти к следующему шагу, пока не выполнен предыдущий.

Выполнение процесса также зависит от доступности информации и не может продолжиться, пока нужная информация не доступна процессу. Разрешение процессу ожидать информации до продолжения — самая важная обязанность менеджера процесса.

В объявлении процесса в таких случаях указано, что необходима более подробная информация. Когда процесс обнаруживает такую ситуацию, он выясняет, какая информация необходима и из какого источника она поступит, а менеджер процессов переводит процесс в неактивное состояние и сохраняет всю информацию, уже полученную процессом.

По достижении состояния ожидания процесс и все данные, которые он использует, удаляются из памяти. Менеджер процессов сохраняет данные в базе данных, запоминает указатель на то место, в котором процесс был остановлен, а затем отвечает за прослушивание входящей информации. Когда новая информация поступает из необходимого источника, менеджер процесса проверяет ее содержание и выбирает нужный экземпляр среди сохраненных процессов. Затем он снова загружает сохраненные данные в память, заполняет процесс всеми данными, которые находились там прежде, до перехода в состояние ожидания, и перезапускает процесс с того места, в котором он был остановлен.

Таким образом, создается впечатление, будто процесс никогда не останавливался, а только задерживался для ожидания части данных.

Менеджер процессов отвечает за запуск процессов, сохранение их при необходимости, ожидание входных данных, а также перезапуск процессов.

Менеджер процессов должен реагировать также в тех случаях, когда процесс не получает необходимую информацию в отведенный для процесса период времени. Это означает, что менеджер процесса должен не только искать входные данные, но и достаточно часто проверять базу данных, хранящую остановленные процессы, и выяснять, не находится ли экземпляр процесса в неактивном состоянии слишком долго.

Модель "запуск–прослушивание–действия"

В самой простой форме у каждого процесса есть три стадии. Процесс запускается менеджером процессов и вызывается приложением. После этого процесс входит в состояние прослушивания, а менеджер процессов выполняет свою работу по сохранению процесса и ждет новой информации перед продолжением.

Когда поступают нужные данные, менеджер процессов оживляет соответствующий процесс, чтобы он выполнял действия по обработке поступившей информации. Модель может иметь более сложную форму, в которой можно принимать решения о том, что прослушивать.

Вы можете сказать, что если в ваших требованиях нет никакой модели вида "запуск–прослушивание–действие", то в действительности нет и процессов, которыми должен управлять менеджер процессов (рис. 1.12).

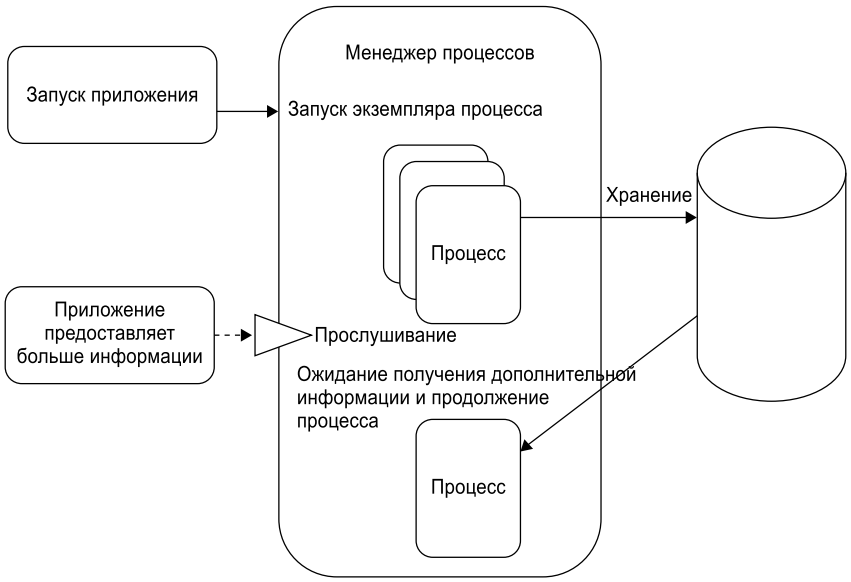


Рис. 1.12. Модель "запуск–прослушивание–действия"

Модели в объявлении последовательности действий

Для объявления процесса нужно выполнить типичную комбинацию действий, которая также может рассматриваться как модель.

Последовательность

Каждый шаг, рассматриваемый как операция, выполняется после предыдущего. Это основная и фундаментальная модель организации процессов. Она допускает нисходящий подход, в котором процесс начинается и продолжается, выполняя следующий шаг в последовательности действий.

Параллельное разбиение

В параллельном разбиении процесс разбивается на несколько ветвей. Каждая ветвь выполняется одновременно с другими и делает свою работу независимо от других ветвей. При параллельном разбиении нет возможности узнать, какая из ветвей будет закончена первой. Возможно, у одной ветви есть только один маленький шаг, который выполнится быстро, а вторая ветвь ждет вмешательства человека. В таком случае время выполнения второй ветви непредсказуемо.

Синхронизация

Данный шаблон часто встречается вместе с параллельным разбиением. Подход параллельного разбиения — ждать завершения всех, прежде чем продолжить. Каждая ветвь делает свою работу, и шаги в последовательности после параллельного разбиения выполняются только тогда, когда все ветви сообщат, что они завершили выполнение задания.

Этот вид параллелизма имеет общие свойства с многозадачными операционными системами, но фактически это менеджер процессов, который планирует выполнение шагов во всех ветвях.

Модель прослушивания

Эту модель также называют исключительным выбором. В ней объявляется ряд ветвей, из которых во время процесса выполняется только одна. Модель основана на механизме, который может решить, какая из ветвей должна быть выполнена.

Этим механизмом является либо задержка на бесконечное время, либо входящее сообщение с информацией, которую ожидает экземпляр процесса. В модели прослушивания описывается несколько условий, которые могут быть выполнены тогда, когда процесс находится в неактивном состоянии. Пребывание в неактивном состоянии означает, что процесс не может сам решить, что делать на следующем шаге. Он будет ждать сигнала перед продолжением.

Поскольку содержимое сигнала важно для принятия этого решения, в модели прослушивания описывается несколько ветвей. Каждая ветвь содержит условие в качестве первого элемента и последовательность сразу после этого условия. Когда поступает сигнал, все условия вычисляются. Когда для ветви найдено соответствие, процесс продолжает выполнение шагов, определенных в данной ветви. Он выполняет только шаги, определенные в его ветви; другие ветви, для которых условие не выполнено, отклоняются, и шаги в этих ветвях никогда не выполняются.

Прослушивание входящих сообщений в течение заранее определенного времени

Во всех ситуациях, где используется модель прослушивания, будет по крайней мере одна ветвь, определяющая задержку. Этот указанный процесс проверяет входящие сообщения на соответствие определенным условиям только в течение заданного промежутка времени. Если это время ожидания истекает прежде, чем хотя бы одно из условий будет выполнено, осуществляется переход на ветвь по задержке. Переход чаще всего выполняет резервный сценарий, сообщающий системе регистрации, что запуск необходимых процедур с возможными ответами для условий в других ветвях не был выполнен.

В данной ветви процесс может выполнить какие-то действия и напомнить другим приложениям и людям, использующим их, о необходимости выполнения некоторых действий. Процесс может оставаться в режиме прослушивания и снова ждать ввода, необходимого для выполнения других действий.

Конвойная модель

В конвойной модели несколько сообщений связаны друг с другом, и для получения результата процессом все они должны быть получены. Это означает, что ни одно сообщение не несет всей информации, необходимой для процесса. В конвоях всегда есть первое сообщение, которое запускает конвой и переводит процесс в состояние ожидания до получения других связанных сообщений. Процесс ждет дальнейших сообщений, которые связаны с первым, и может продолжиться только после получения всей информации. Наличие всех сообщений — именно то, чего ждет процесс. Процесс нуждается либо во множественных сообщениях, чтобы работать с набором как с одной большой частью связанных данных, либо, по крайней мере, во многих сообщениях из различных источников, чтобы принять решение о продолжении. Данные для ответа на запрос разбиваются на части, которые передаются процессу одна за другой.

Причина такого разделения на части может быть технической, связанной с ограничением памяти, заставляющим разбить данные на несколько частей. Но может быть и функциональной, если входные данные для процесса представляют собой агрегированные данные, исходящие из различных приложений, которые могут посылать только свои собственные данные. Агрегирование данных из различных источников — именно то, для чего необходимы бизнес-процессы.

Ниже приведены некоторые примеры.

- Процесс выполняет инструкцию и получает несколько сообщений. Эта инструкция содержит упорядоченные строки, потому что приложение-отправитель не может послать их в требуемом порядке из-за технических ограничений.
- Процесс ищет поставщика и ожидает, пока по крайней мере три ценовых котировки не будут получены перед выбором поставщика.

Последовательный конвой

В последовательном конвое у различных сообщений есть предопределенная последовательность. Для получения результата процессу необходимо несколько сообщений, но он может получить только одно сообщение за один раз. Это то сообщение, которое он ожидает согласно месту в последовательности предыдущего сообщения. Такой случай может иметь место тогда, когда данные в сообщениях необходимы для принятия решения о том, какого сообщения ждать следующим. Если информация в сообщении указывает некоторый приоритет обработки сообщений, необходим последовательный конвой.

Параллельный конвой

В параллельном конвое у сообщений нет строгой последовательности, и процесс может обрабатывать сообщения в любой очередности. Но как и в параллельной модели, процесс продолжается только после того, как получит все сообщения конвоя. В параллельном конвое известно либо число сообщений, которые будут ожидаться перед началом ввода сообщений, либо свойство сообщений, указывающее на то, является ли оно последним сообщением в конвое или нет.

2

Соглашения о службах и соглашения о данных

В ЭТОЙ ГЛАВЕ...

- Описание соглашений
- Реализация соглашений о службах
- Реализация соглашений о данных
- Использование различных преобразователей
- Реализация соглашений о сообщениях

Соглашения в целом очень важны в нашей повседневной жизни. Благодаря соглашениям каждая из сторон знает, чем окупятся ее усилия. Если вы поставили свою подпись в конце 17 страниц деловой документации или заключили устное соглашение, ожидания считаются определенными и согласованными. Соглашения должны быть четко определены и понятны и не должны оставлять свободу для интерпретации.

Ясно одно: в служебно-ориентированной среде, в которой различные субъекты (компоненты программного обеспечения) взаимодействуют друг с другом и используют различные технологии и платформы, крайне необходимо иметь четкое определение всего, что при этом происходит.

WCF (Windows Communication Foundation) использует концепцию соглашений для определения службы и его функционирования в целом, для описания передаваемых данных и, если необходимо, непосредственного определения SOAP-сообщений, которыми обмениваются клиент и служба. WCF использует WSDL и XSD для доставки метаданных службы.

В WCF есть три типа соглашений: о службах, данных и сообщениях.

Каждое из этих соглашений определяет свои характеристики. Какие из них необходимы, зависит от вашей архитектуры. Наиболее часто используется соглашение о службе, которое может использоваться почти в любом сценарии. Соглашение о службе также определяет конечную точку в цепи. Как вы знаете, конечной точкой службы является адресуемый элемент, который обычно используется клиентом для отправки и получения сообщений.

Соглашения в контексте SOA обеспечивают необходимые метаданные для связи со службой, — они описывают такие вещи, как типы данных, операции, шаблоны обмена сообщениями, а также используемый транспортный протокол. Соглашения, как правило, поставляются на стандартизированной платформе и в независимом от языка формате XML, что позволяет различным сторонам использовать и интерпретировать их определение. В WCF метаданные службы описываются в основном документом WSDL (Web Services Description Language). Более подробная информация доступна по адресу `http://www.w3.org/TR/wsdl`. Этот стандарт позволяет поставщикам программного обеспечения, таким как Microsoft, Sun и IBM, разрабатывать инструменты, помогающие разработчикам создавать и использовать эти соглашения.

Соглашения о службах

Соглашения о службах определяют функциональные возможности, которые ваш служба предлагает внешнему миру. Функциональные возможности выражаются в форме операций службы. В мире SOA вы обычно обмениваетесь набором SOAP-сообщений, и схемы этих сообщений определяются в соглашении о службе. Соглашение о службе также определяет шаблон обмена сообщениями (запрос/ответ, односторонний, дуплекс) для каждой операции службы.

Соглашения о данных

Обычно вы передаете один или несколько аргументов операциям службы и ожидаете возврата значения. Структура аргументов сообщения определяется в соглашении о данных в виде схемы XSD, которая является частью документа WSDL. Соглашение о данных определяет структуру и содержание информации, которой обмениваются клиент и служба.

Если вы используете собственные объекты и сложные типы в качестве аргументов метода или возвращаемых значений, то должны информировать выполняющуюся программу, как преобразовать эти типы в поток XML с помощью атрибута `DataContract`.

В зависимости от детализации аргументов вашего метода можно использовать только простые .NET типы как аргументы метода. В этом случае вам не нужно беспокоиться о преобразовании, потому что инфраструктура обмена сообщениями уже знает, как обрабатывать эти простые .NET типы.

Соглашения о сообщениях

Соглашение о сообщениях — это свойство, которое дает дополнительные возможности управления заголовком или телом SOAP. Соглашение о сообщениях используется для непосредственного взаимодействия с сообщением SOAP (заголовком и телом), а не только с телом метода, как и в случае, если вы используете соглашение о данных, а не о сообщении.

Соглашение и код

Разработчику .NET программного обеспечения для службы далеко не каждый день приходится писать файлы метаданных в виде WSDL, — ему, как правило, нужно создавать интерфейсы и классы, писать методы, реализовывать бизнес-логику, а затем писать тесты модулей, чтобы проверить логику. Чтобы помочь разработчику программного обеспечения создавать эти искусственные объекты метаданных из исходных кодов без коренного изменения программного кода, WCF предлагает различные инструменты, атрибуты и классы, которые более подробно будут рассмотрены далее.

При использовании WCF для реализации архитектуры SOA можно выбрать различные подходы в зависимости от требований — автоматическое создание этих документов метаданных из заготовок на основе кода. Можно использовать атрибуты .NET, параметры конфигурации, а также непосредственно предоставляемые WCF средства API для полного управления генерацией метаданных.

Но как обрабатывать метаданные, полученные от клиентов, и взаимодействовать со службой? Для завершения картины предположим, что у вас уже сформирована спецификация службы в виде документа WSDL. Далее будем считать, что служба запущена и работает. Чтобы взаимодействовать со службой, необходимо создать сообщение XML в нужном формате, послать сообщение в конечную точку службы, ждать ответного сообщения и, наконец, извлечь данные из документа XML. Благодаря стандарту WSDL и инструментам, таким как `svcutil.exe` и VS 2010, очень легко создать исходный код (прокси), который имитирует поведение службы и обрабатывает все сообщения и детали преобразований в последовательную форму.

Как упоминалось ранее, существуют различные подходы к контролю над выполнением программ. Вам нужно знать, какие части кода должны быть отражены в виде метаданных в WSDL, а не точную спецификацию WSDL. Полезно знать также некоторые основные элементы WSDL, чтобы хорошо понимать, что происходит “за кулисами”, и интерпретировать полученные документы метаданных. Чтобы понять основные идеи взаимодействия элементов в документе WSDL и вашего кода, начните с простого примера, чтобы увидеть, какие фрагменты кода приводят к какому элементу в документе WSDL.

Прокат автомобилей — пример реализации

Пока в целом были объяснены автоматическая генерация метаданных и использование прокси. Теперь настало время увидеть все эти понятия в действии.

Чтобы дать вам краткий и компактный обзор того, как создать соглашение о службе, показана упрощенная реализация вымышленной службы проката автомобилей с подробными пояснениями кода листингов. Если вы выберете подход на основе описания службы, то, скорее всего, начнете с определения документа WSDL, а не с написания кода.

При использовании основанного на коде подхода нужно начать с определения интерфейса .NET и дополнить контракт службы атрибутом `[ServiceContract]`, а ваши методы — атрибутом `[OperationContract]` из пространства имен `System.ServiceModel`.

Шаг 1. Определение контракта службы

Определения контрактов службы, как показано в листинге 2.1, должны находиться в своем собственном модуле, поскольку есть несколько преимуществ в использовании отдельной библиотеки классов для интерфейса и контрактов службы, таких как управление версиями, отделение абстрактных определений от конкретной реализации и распространение.

Листинг 2.1. Определение контрактов службы


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
  [ServiceContract()]
  public interface ICarRentalService
  {
   [OperationContract]
   double CalculatePrice(DateTime pickupDate, DateTime returnDate,
   string pickupLocation, string vehiclePreference);
  }
}
```

Как видно из листинга 2.1, пример интерфейса службы достаточно прост и содержит всего один метод, `CalculatePrice`, с четырьмя аргументами и возвращаемым значением типа `double`.

Чтобы определить контракт службы, примените атрибут `[ServiceContract]` для определения интерфейса и атрибут `[OperationContract]` для каждого метода, который следует включить в качестве операции службы в документ WSDL. Если интерфейс содержит метод без атрибута `[OperationContract]`, такой метод не будет включен в созданные метаданные службы. Это также называется *моделью включения*.

Можно также аннотировать класс — конкретную реализацию службы — атрибутом `[ServiceContract]`, а его методы — атрибутом `[OperationContract]`. Тем не менее есть несколько причин, почему делать это не рекомендуется. Одна из них заключается в том, что вы будете смешивать "внешний мир" (соглашение о службе) с "внутренним миром" (код реализации), а это нарушает правило о явных границах. Всегда необходимо отделять абстрактное определение службы от конкретной реализации, — только в этом случае можно создать производительный код.

Дополнительно можно создать различные реализации того же соглашения о службе и развернуть их в различных конечных точках службы, если применить данный атрибут на уровне интерфейса.

После компиляции исходного кода получится собранный модуль `Wrox.CarRentalService.Contracts.dll`, который содержит определения соглашений об интерфейсе и службе.

Шаг 2. Извлечение метаданных службы

Чтобы лучше понимать, как каждая часть определения интерфейса соотносится с документом WSDL, в первую очередь создается документ WSDL.

Если служба предоставляет Metadata Exchange Endpoint (MEX), можно сразу перейти к этой конечной точке и посмотреть порожденные метаданные.

Если у вас нет MEX, можно с помощью `svcutil.exe` сгенерировать файл WSDL. Инструмент ServiceModel Metadata Utility Tool (`svcutil.exe`), находящийся в папке установки Windows SDK, предоставляет ряд функциональных возможностей.

- Генерация кода из запущенных служб или статических документов метаданных.
- Экспорт метаданных из скомпилированного кода.
- Проверка скомпилированного кода службы.

- Загрузка метаданных документа из запущенной службы.
- Генерация кода сериализации.

Используйте команду `svcutil.exe Wrox.CarRentalService.Contracts.dll` в командной строке Visual Studio, чтобы получить три файла, приведенных в табл. 2.1.

Таблица 2.1. Файлы WSDL и XSD

Имя файла	Описание
`tempuri.org.wsdl` (см. листинг 2.2)	Документ WSDL со ссылкой на файл XSD, определения сообщений, типы портов служебных операций
`tempuri.org.xsd` (см. листинг 2.3)	Схема XSD для атрибутов метода
`schemas.microsoft.com.2003.10.Serialization.xsd` (см. листинг 2.4)	Стандартная схема XSD для простых типов .NET

Дополнительная информация о схемах содержится на сайте W3C по адресу `http://www.w3.org/XML/Schema`.

Листинг 2.2. WSDL

Доступно для загрузки на Wrox.com

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:tns="http://tempuri.org/"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
xmlns:wsa10="http://www.w3.org/2005/08/addressing"
xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
targetNamespace="http://tempuri.org/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://tempuri.org/Imports">
      <xsd:import namespace="http://tempuri.org/" />
      <xsd:import namespace="http://schemas.microsoft.com/2003/10/
Serialization/" />
  </xsd:schema>
  </wsdl:types>
  <wsdl:message name="ICarRentalService_CalculatePrice_InputMessage">
    <wsdl:part name="parameters" element="tns:CalculatePrice" />
  </wsdl:message>
```

```
  <wsdl:message name="ICarRentalService_CalculatePrice_OutputMessage">
    <wsdl:part name="parameters" element="tns:CalculatePriceResponse" />
  </wsdl:message>
  <wsdl:portType name="ICarRentalService">
    <wsdl:operation name="CalculatePrice">
      <wsdl:input wsaw:Action="http://tempuri.org/ICarRentalService/CalculatePrice"
message="tns:ICarRentalService_CalculatePrice_InputMessage" />
      <wsdl:output wsaw:Action="http://tempuri.org/ICarRentalService/
CalculatePriceResponse" message=
"tns:ICarRentalService_CalculatePrice_OutputMessage" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="DefaultBinding_ICarRentalService"
type="tns:ICarRentalService">
    <soap:binding transport="http://schemas.xmlsoap.org/soa
    <wsdl:operation name="CalculatePrice">
      <soap:operation soapAction="http://tempuri.org/ICarRe
CalculatePrice" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  </wsdl:definitions>
```

Листинг 2.3. XSD-файл


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://tempuri.org/" elementFormDefault="qualified"
targetNamespace="http://tempuri.org/" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CalculatePrice">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="pickupDate" type="xs:dateTime" />
        <xs:element minOccurs="0" name="returnDate" type="xs:dateTime" />
        <xs:element minOccurs="0" name="pickupLocation" nillable="true"
type="xs:string" />
        <xs:element minOccurs="0" name="vehiclePreference"
nillable="true" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CalculatePriceResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="CalculatePriceResult" type="xs:double" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Листинг 2.4. `schemas.microsoft.com.2003.10.Serialization`


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns=http://schemas.microsoft.com/2003/10/Serialization/
 attributeFormDefault="qualified" elementFormDefault="qualified"
targetNamespace="http://schemas.microsoft.com/2003/10/Serialization/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="anyType" nillable="true" type="xs:anyType" />
  <xs:element name="anyURI" nillable="true" type="xs:anyURI" />
  <xs:element name="base64Binary" nillable="true"
type="xs:base64Binary" />
  <xs:element name="boolean" nillable="true" type="xs:boolean" />
  <xs:element name="byte" nillable="true" type="xs:byte" />
  <xs:element name="dateTime" nillable="true" type="xs:dateTime" />
  <xs:element name="decimal" nillable="true" type="xs:decimal" />
  <xs:element name="double" nillable="true" type="xs:double" />
  <xs:element name="float" nillable="true" type="xs:float" />
  <xs:element name="int" nillable="true" type="xs:int" />
  <xs:element name="long" nillable="true" type="xs:long" />
  <xs:element name="QName" nillable="true" type="xs:QName" />
  <xs:element name="short" nillable="true" type="xs:short" />
  <xs:element name="string" nillable="true" type="xs:string" />
  <xs:element name="unsignedByte" nillable="true"
type="xs:unsignedByte" />
  <xs:element name="unsignedInt" nillable="true"
type="xs:unsignedInt" />
  <xs:element name="unsignedLong" nillable="true"
type="xs:unsignedLong" />
  <xs:element name="unsignedShort" nillable="true"
type="xs:unsignedShort" />
  <xs:element name="char" nillable="true"
type="tns:char" />
  <xs:simpleType name="char">
    <xs:restriction base="xs:int" />
  </xs:simpleType>
  <xs:element name="duration" nillable="true"
type="tns:duration" />
  <xs:simpleType name="duration">
    <xs:restriction base="xs:duration">
      <xs:pattern value=
"\-?P(\d*D)?(T(\d*H)?(\d*M)?(\d*(\.\d*)?S)?)?" />
      <xs:minInclusive value="-P10675199DT2H48M5.4775808S" />
      <xs:maxInclusive value="P10675199DT2H48M5.4775807S" />
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="guid" nillable="true" type="tns:guid" />
  <xs:simpleType name="guid">
    <xs:restriction base="xs:string">
      <xs:pattern value=
"[\da-fA-F]{8}-[\da-fA-F]{4}-[\da-fA-F]{4}-
[\da-fA-F]{4}-[\da-fA-F]{12}" />
    </xs:restriction>
  </xs:simpleType>
  <xs:attribute name="FactoryType" type="xs:QName" />
```

```
  <xs:attribute name="Id" type="xs:ID" />
  <xs:attribute name="Ref" type="xs:IDREF" />
</xs:schema>
```

Как видно из листингов 2.2 (WSDL) и 2.3 (XSD), различные части XSD и WSDL создаются непосредственно из исходного кода. Чтобы изменить методику присваивания имен по умолчанию и получить дополнительные возможности управления, можно использовать члены атрибутов `[ServiceContract]` и `[OperationContract]`.

В табл. 2.2 показано соответствие между кодом и документами WSDL.

Таблица 2.2. Соответствие между исходным кодом и элементами WSDL

Элемент WSDL	Исходный код
`Message name: ICarRentalService_CalculatePrice`	Имя интерфейса + имя метода + ввод / вывод
`portType name: ICarRentalService`	Имя интерфейса
`Operation name: CalculatePrice`	Имя метода

До сих пор мы создавали метаданные из кода путем определения интерфейса, добавления некоторых методов к нему и дополнения имени интерфейса атрибутом `[ServiceContract]` и методов пользователя — атрибутом `[OperationContract]`. Инфраструктура WCF использует отражение этих данных в созданном документе WSDL.

Шаг 3. Реализация службы

Для завершения служебной части нужна одна или несколько конкретных реализаций интерфейса службы. Конкретные реализации службы рекомендуется помещать в отдельные библиотеки классов. Как показано в листинге 2.5, в реализации кода службы нет необходимости использовать специфические элементы WCF.

Использование атрибутов `[ServiceBehavior]` и `[OperationBehavior]` можно увидеть в коде реализации. Поведение в целом представляет собой внутренние детали реализации и, следовательно, не включается в файл WSDL.

Листинг 2.5. Реализация службы


```
using System;
using Wrox.CarRentalService.Contracts;

namespace Wrox.CarRentalService.Implementations.Europe
{
    public class CarRentalService: ICarRentalService
    {
        public double CalculatePrice
                      (
                      DateTime pickupDate,
                      DateTime returnDate,
                      string pickupLocation,
                      string vehiclePreference
                      )
        {
            // вызов внутренней бизнес-логики
            Random r = new Random(DateTime.Now.Millisecond);
```

```
            return r.NextDouble()*500;
        }
    }
}
```

Шаг 4. Клиент

Метаданные можно использовать для создания необходимого кода клиента. Для этого клиент использует `svcutil.exe` или Add Service Reference в оболочке Visual Studio для получения прокси, который имитирует функционирование службы. (Подробно об этом — в главе 5.) Листинг 2.6 демонстрирует использование созданного прокси в клиентском приложении; листинг 2.7 содержит сообщение SOAP с запросом, а листинг 2.8 — ответное сообщение.

Листинг 2.6. Код клиента


```
using System;
using Wrox.CarRentalService.ConsoleClient.CarRentalProxy;

namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (CarRentalServiceClient
                   carRentalClient = new CarRentalServiceClient())
            {
                double price = carRentalClient.CalculatePrice
                       (
                        DateTime.Now, DateTime.Now.AddDays(5),
                        "Graz", "Pickup"
                       );
                Console.WriteLine("Price {0}",price );
            }
        }
    }
}
```

Листинг 2.7. Сообщение SOAP


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <CalculatePrice xmlns="http://tempuri.org/">
   <pickupDate>  2010-01-09T19:15:25.2392813+01:00  </pickupDate>
   <returnDate>  2010-01-14T19:15:25.2402578+01:00  </returnDate>
   <pickupLocation>  Graz  </pickupLocation>
   <vehiclePreference>  Pickup  </vehiclePreference>
  </CalculatePrice>
 </s:Body>
</s:Envelope>
```

Листинг 2.8. Ответное сообщение SOAP


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <CalculatePriceResponse xmlns="http://tempuri.org/">
   <CalculatePriceResult>  357.7265724808567  </CalculatePriceResult>
  </CalculatePriceResponse>
 </s:Body>
</s:Envelope>
```

[ServiceContract] и [OperationContract]

Автоматическая генерация документа WSDL иногда требует дальнейшего усовершенствования, — для службы, чтобы предотвратить конфликты имен с другими службами, нужно хотя бы обеспечить осмысленное пространство имен. Иногда применяются некоторые внутренние стили кодирования и шаблоны именования для интерфейсов, классов и методов, которые не разрешены или не рекомендованы в описании службы, или если вы не хотите, чтобы внутренние имена были видимы из внешнего мира.

В целях дальнейшего расширения возможностей управления полученными документами WSDL в табл. 2.3 (из MSDN) приведено подробное описание членов `[ServiceContract]` и `[OperationContract]`.

Таблица 2.3. Члены `[ServiceContract]`

Свойство	Описание
`CallbackContract`	Получает или задает тип обратного вызова, когда соглашение является дуплексной моделью
`ConfigurationName`	Получает или задает имя, используемое для поиска службы в файле конфигурации приложения
`HasProtectionLevel`	Получает значение, указывающее, назначен ли уровень защиты для членов класса
`Name`	Получает или задает имя элемента `<portType>` в WSDL
`Namespace`	Получает или задает пространство имен элемента `<portType>` в WSDL
`ProtectionLevel`	Указывает, обязательно ли соединение для соглашения должно поддерживать значение свойства `ProtectionLevel`
`SessionMode`	Получает или задает, разрешены ли сессии, не разрешены, или требуется

То же самое применяется к атрибуту `[OperationContract]`, как и к атрибуту `[ServiceContract]`, — иногда нужны дополнительные средства управления процессом генерации документа WSDL.

См. таблицу 2.4.

Таблица 2.4. Члены `[OperationContract]`

Параметр	Описание
`Action`	Определяет действие, которое однозначно идентифицирует эту операцию. WCF отправляет сообщения с запросами методам на основе их действий
`AsyncPattern`	Указывает, что операция осуществляется или может быть вызвана асинхронно с помощью пары `Begin/End`

Окончание табл. 2.4

Параметр	Описание
`HasProtectionLevel`	Указывает, что значение `ProtectionLevel` было задано явно
`IsOneWay`	Указывает, что операция состоит только из одного входного сообщения. Операция не генерирует исходящее сообщение
`IsInitiating`	Определяет, может ли эта операция быть начальной в сессии
`ProtectionLevel`	Определяет уровень безопасности, необходимый для выполнения операции
`IsTerminating`	Определяет, будет ли WCF пытаться прекратить текущую сессию после завершения операции

В листинге 2.9 показано, как использовать некоторые общие атрибуты. Другие атрибуты описаны в главе 3.

Листинг 2.9. Атрибуты `[ServiceContract]` и `[OperationContract]`


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(Namespace="http://wrox/CarRentalService/2009/10",
                     Name="RentalService")]
    public interface ICarRentalService
    {
        [OperationContract(Name = "GetPrice")]
        double CalculatePrice(DateTime pickupDate, DateTime returnDate,
        string pickupLocation, string vehiclePreference);

        [OperationContract( Name = "GetPriceOverloaded")]
        double CalculatePrice(string pickupLocation,
        string vehiclePreference);

        [OperationContract(IsOneWay=true,
        ProtectionLevel=System.Net.Security.ProtectionLevel.None )]
        void UpdatePrice(string vehicleId, double newPrice);
    }
}
```

Результаты использования этих атрибутов отражены во фрагменте WSDL, приведенном в листинге 2.10.

Листинг 2.10. Документ WSDL


```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions . . ..
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace=
"http://wrox/CarRentalService/2009/10"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:message name="RentalService_GetPrice_InputMessage">
    <wsdl:part name="parameters" element="tns:GetPrice" />
  </wsdl:message>
```

```
  <wsdl:message name="RentalService_GetPrice_OutputMessage">
    <wsdl:part name="parameters" element="tns:GetPriceResponse" />
  </wsdl:message>
  <wsdl:message name="RentalService_GetPriceOverloaded_InputMessage">
    <wsdl:part name="parameters" element="tns:GetPriceOverloaded" />
  </wsdl:message>
  <wsdl:message name="RentalService_GetPriceOverloaded_OutputMessage">
  <wsdl:part name="parameters" element="tns:GetPriceOverloadedResponse" />
  </wsdl:message>
  <wsdl:message name="RentalService_UpdatePrice_InputMessage">
    <wsdl:part name="parameters" element="tns:UpdatePrice" />
  </wsdl:message>
  <wsdl:portType name="RentalService">
    <wsdl:operation name="GetPrice">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPrice"
 message="tns:RentalService_GetPrice_InputMessage" />
      <wsdl:output wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPriceResponse"
 message="tns:RentalService_GetPrice_OutputMessage" />
    </wsdl:operation>
    <wsdl:operation name="GetPriceOverloaded">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPriceOverloaded"
 message="tns:RentalService_GetPriceOverloaded_InputMessage" />
      <wsdl:output wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService
/GetPriceOverloadedResponse"
message="tns:RentalService_GetPriceOverloaded_OutputMessage" />
    </wsdl:operation>
    <wsdl:operation name="UpdatePrice">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/UpdatePrice"
 message="tns:RentalService_UpdatePrice_InputMessage" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Соглашения о данных

Предыдущие примеры продемонстрировали использование атрибутов `[OperationContract]` и `[ServiceContract]`. Атрибут `[OperationContract]`, по существу, определяет, какие операции предоставляются провайдером службы.

Передаваемые данные зависят от индивидуальных параметров передачи и возвращаемого типа данных. Конкретное значение параметра, как правило, передается от объекта .NET в памяти.

Этот объект преобразуется в соответствующую форму и вкладывается в сообщение SOAP. С другой стороны, параметры извлекаются из сообщения SOAP и предоставляются в виде объекта .NET. Это преобразование осуществляется специальными классами сериализации.

WCF поддерживает различные типы сериализации и десериализации, причем некоторые преобразователи были специально разработаны для WCF, хотя некоторые из них существовали в рамках .NET 1.0 и .NET 2.0 и также могут быть использованы с WCF.

WCF использует класс `DataContractSerializer` из пространства имен `System.Runtime.Serialization` в стандартной комплектации для сериализации или десериализации своих объектов. Этот сериализатор был специально разработан для WCF и является особенно быстрым, эффективным и мощным, что определяет его как предпочтительный выбор. Однако, если ваши требования превышают возможности `DataContractSerializer`, можно вернуться к классической сериализации XML.

`DataContractSerializer` поддерживает различные типы данных, включая приведенные ниже.

- Примитивные типы данных.
- Типы данных с атрибутами `[DataContract]`.
- Классы, которые отмечены как сериализуемые.
- Классы, реализующие интерфейс `IXmlSerializable` из пространства имен `System.Xml.Serialization`.
- Перечисления, коллекции и универсальные коллекции.

В предыдущих примерах не было необходимости в дополнительных уровнях сериализации, потому что примитивные типы данных уже сериализуемы. Однако, если приходится работать со сложными типами данных, такими как параметры методов или возвращаемые значения, нужно использовать соответствующие атрибуты (`[DataContract]` и `[DataMember]`), чтобы определить, как эти данные преобразуются в последовательную форму.

Эти два атрибута из пространства имен `System.Runtime.Serialization`, которые конструируются во время разработки, объясняют классу `DataContractSerializer`, как сериализировать специальные типы. Поскольку структуры передаваемых данных также принадлежат соглашению, конечно, применение атрибутов `[DataContract]` и `[DataMember]` непосредственно влияет на подготовленный документ WSDL. Определения данных включены в соглашение (WSDL) в формате, который не зависит от платформы и языков программирования, а именно XSD. Таким образом, клиент и сервер не могут согласиться на обмен типами данных .NET, но совместимы с учетом нейтральной схемы XML определения формата.

С этой целью в листинге 2.11 приведен простой пример, который не включают использование атрибутов `[DataContract]` и `[DataMember]`, а создает документ WSDL, показанный в листинге 2.12.

Листинг 2.11. Операции службы с простыми типами


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(
             Namespace = "http://wrox/CarRentalService/2009/10",
             Name = "RentalService")]

    public interface ICarRentalService
    {
        [OperationContract]
        double CheckAvgPricePerDay(string carType);
    }
}
```

Листинг 2.12. Документ WSDL с простыми типами


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://wrox/CarRentalService/2009/10" elementFormDefault=
"qualified"
targetNamespace="http://wrox/CarRentalService/2009/10"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CheckAvgPricePerDay">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="carType"
nillable="true" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CheckAvgPricePerDayResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0"
name="CheckAvgPricePerDayResult"
type="xs:double" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

В примере, приведенном в листинге 2.11, ни атрибут `[Serializable]`, ни атрибут `[DataContract]` не использовался с соответствующим атрибутом `[DataMember]`. Это возможно потому, что класс `DataContractSerializer` может работать с простыми .NET типами данных без каких-либо мер предосторожности.

Однако в большинстве случаев в качестве содержимого методов в ориентированной на сообщения архитектуре используются не простые типы данных, вместо этого внимание должно быть уделено структуре передаваемых данных и представления их в формате соглашения.

В этом случае можно использовать атрибуты `[DataContract]` и `[DataMember]`, чтобы сообщить классу `DataContractSerializer`, как должны быть преобразованы его объекты и в какую форму.

Если в операциях используются сложные типы данных без указания классу `DataContractSerializer`, как они должны быть обработаны, все общедоступные члены класса сериализируются автоматически. Это позволяет провести сериализацию POCO (простых объектов CLR) без дополнительных изменений в коде. Вы можете исключить отдельные члены с помощью атрибута `[IgnoreDataMember]`. Листинг 2.13 показывает соглашение о службе, в котором метод называется `CalculatePrice`, входной параметр имеет тип `PriceCalculationRequest`, а тип возвращаемого значения будет `PriceCalculationResponse`.

Листинг 2.13. Контракт службы со сложными типами имен


```
namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(
              Namespace = "http://wrox/CarRentalService/2009/10",
              Name = "RentalService")]
    public interface ICarRentalService
```

```
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
            PriceCalculationRequest request);
        }
    }
}
```

Листинги 2.14 и 2.15 демонстрируют использование POCO, атрибута `[IgnoreDataMember]` и результирующий документ WSDL.

Листинг 2.14. Объекты POCO и `[IgnoreDataMember]`


```
using System;
using System.Runtime.Serialization;
using System.Xml.Serialization;

namespace Wrox.CarRentalService.Contracts
{
    public class PriceCalculationRequest
    {
        public DateTime PickupDate { get; set; }
        public DateTime ReturnDate { get; set; }
        public string PickupLocation { get; set; }
        public string ReturnLocation { get; set; }
        private string VehicleType { get; set; }
        [IgnoreDataMember]
        public string Color { get; set; }
    }
}
```

Листинг 2.15. Документ WSDL


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
elementFormDefault="qualified"
targetNamespace="http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="PriceCalculationRequest">
    <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="PriceCalculationRequest"
nillable="true" type="tns:PriceCalculationRequest" />
  <xs:complexType name="PriceCalculationResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="Flag" type="xs:int" />
```

```
      <xs:element minOccurs="0" name="Price" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="PriceCalculationResponse"
nillable="true" type="tns:PriceCalculationResponse" />
</xs:schema>
```

Однако в очень редких случаях можно сделать общедоступными свои бизнес-объекты, включая все общедоступные свойства.

Важно определить соглашения явно в сценарии SOA и обеспечить контроль над именами, в том числе пространством имен ваших элементов, порядком поступления отдельных атрибутов, и над тем, насколько это необходимо.

Следовательно, соглашение о данных и соглашение о службе вместе служат для формирования формального соглашения между службой и клиентом и предоставляют подробное описание структуры данных, которыми обмениваются клиент и служба.

`DataContractSerializer` использует атрибуты `[DataContract]` и `[DataMember]` для управления точным процессом сериализации. `[DataContract]` является моделью подключения, что означает, что сериализируются только свойства или переменные-члены, которым был явно назначен атрибут `[DataMember]`.

Видимость свойств не имеет никакого значения. В листинге 2.16 демонстрируется использование атрибутов `[DataContract]` и `[DataMember]`. В листинге 2.17 приведена получающаяся часть XSD.

Листинг 2.16. Атрибут `[DataContract]` и `[DataMember]`


```
using System;
using System.Runtime.Serialization;
using System.Xml.Serialization;

namespace Wrox.CarRentalService.Contract
{
    [DataContract]
    public class PriceCalculationRequest
    {
        [DataMember]
        public DateTime PickupDate { get; set; }
        [DataMember]
        public DateTime ReturnDate { get; set; }
        [DataMember]
        public string PickupLocation { get; set; }
        [DataMember]
        public string ReturnLocation { get; set; }
        public string Color { get; set; }
    }
}
```

Листинг 2.17. Часть XSD


```
<xs:complexType name="PriceCalculationRequest">
    <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
```

```
nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
```

Характеристики схемы, которые автоматически генерируются WCF, включают следующее.

- Пространства имен: `http://schemas.datacontract.org/2004/07/ + CLR` и `http://schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts`.
- Атрибуты расположены в алфавитном порядке
- `<xs:element minOccurs="0" name= PickupDate "`
- `<xs:element minOccurs="0" name="PickupLocation "`
- Значение атрибута `MinOccurs` установлено равным 0, что означает, что этот атрибут не является обязательным
- Примитивные типы данных CLR автоматически отображаются в типы данных XSD. `DateTime PickupDate` становится `xs: dateTime`

Соглашения о данных в деталях

Как вы видели из предыдущих примеров, классам данных, как правило, присваивается атрибут `[DataContract]`, а отдельным переменным-членам — атрибут `[DataMember]`.

Листинг 2.18 демонстрирует дополнительные возможности управления порождением схемы с помощью атрибутов `[DataContract]` и `[DataMember]`.

Листинг 2.18. Атрибуты `[DataContract]` и `[DataMember]`


```
[DataContract(
Name="PriceCalculationRequest", Namespace=
"http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts")
]
public class PriceReq
{
   [DataMember(Name="PickupDate",Order=1, IsRequired=true )]
   private DateTime FromDate { get; set; }
   [DataMember(Name = "ReturnDate", Order = 3)]
   public DateTime ToDate{ get; set; }
   [DataMember( Order = 2)]
   public string PickupLocation { get; set; }
   [DataMember(Order = 4)]
   public string ReturnLocation { get; set; }
   public string CarType { get; set; }
}
```

Хотя первоначальное определение класса `PriceCalculationRequest` претерпело существенные изменения, проблемы при сериализации экземпляров по сравнению с подготовленной ранее схемой не возникают. Атрибуты `[DataContract]` и `[DataMember]` создают набор данных XML, который согласуется с соглашением (XSD).

Чтобы проиллюстрировать это, обратимся к вновь подготовленной схеме в листинге 2.19, которая совместима с первоначальной схемой.

Листинг 2.19. Эквивалент XSD


```
<xs:complexType name="PriceCalculationRequest">
   <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
           nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
           nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
```

Следующие наблюдения были сделаны при использовании атрибутов `[DataContract]` и `[DataMember]` из пространства имен `System.Runtime.Serialization`.

- Пространство имен по умолчанию может быть переопределено свойством `namespace`.
- Свойство `name` атрибута `[DataContract]` служит для определения имени сложного типа в XSD.
- Модификатор доступа к переменной-члену класса игнорируется.
- `CarType` не является элементом в модели подключения — сериализируются только переменные-члены, которым явно приписан атрибут `[DataMember]`.
- Если свойство `IsRequired = True`, атрибут далее является обязательным. Если этот атрибут не найден в процессе десериализации, возникает ошибка.
- Порядок может быть определен явно с помощью свойства `order`.
- Свойство `name` атрибута `[DataMember]` используется для определения имени элемента XSD.

Свойства атрибутов `[DataContract]` и `[DataMember]` в основном играют важную роль в управлении версиями или в сценарии взаимодействия. Если, например, нужно сохранить установленную по умолчанию схему XML, желательно избежать каких-либо изменений, за исключением разве что лишь незначительных, в классах .NET, можно попытаться создать совместимые с соглашением о данных признаки и свойства, которые упоминались ранее.

Атрибут `KnownTypes`

В объектно-ориентированном программирования часто используются ссылки на унаследованный, а не на базовый класс. Эта концепция полиморфизма используется весьма часто, хотя, если не будут приняты особые меры предосторожности, в WCF при таком подходе возникает ошибка.

Спецификация метода обычно содержит только базовый класс и, следовательно, определение унаследованных классов тоже не включается в документ WSDL. Естественно, это также означает, что унаследованный класс неизвестен для сгенерированного кода прокси и поэтому может не десериализироваться.

Та же проблема возникает, например, при использовании параметра, представляющего собой неуниверсальную коллекцию (класс), как показано в листинге 2.20.

Листинг 2.20. Аргумент ответного сообщения базового класса и `ArrayList`


```
namespace Wrox.CarRentalService.Contracts
{

    [ServiceContract(
       Namespace = "http://wrox/CarRentalService/2009/10",
       Name = "RentalService")]
    [ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
    public interface ICarRentalService
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
            PriceCalculationRequest request
        );

        [OperationContract]
        System.Collections.ArrayList GetPrices();
    }
```

Однако конкретная реализация метода в листинге 2.20 не использует базовый класс `PriceCalculationResponse` в качестве возвращаемого значения, она использует унаследованную версию, которая называется `PriceCalculationResponseDetailed`. В листинге 2.21 показан унаследованный класс и конкретная реализация службы.

Листинг 2.21. Унаследованный класс и реализация службы


```
[DataContract]
public class PriceCalculationResponseDetailed
                : PriceCalculationResponse
{
  [DataMember]
  public string  Currency { get; set; }
}


public PriceCalculationResponse
        CalculatePrice(PriceCalculationRequest  request)
{
   PriceCalculationResponseDetailed resp = null;
   reps=new PriceCalculationResponseDetailed();
   resp.Price = 120;
   resp.Currency = "euro";
   return resp;
}
```

Теперь, если прокси-клиент вызывает метод `CalculatePrice`, возвращаемое значение не ожидается, и поэтому возникает следующая ошибка:

```
There was an error while trying to serialize parameter http://wrox/
CarRentalService/2009/10:CalculatePriceResult. The InnerException message
was 'Type ' Wrox.CarRentalService.Contracts.PriceCalculationResponse-
Detailed ' with data contract name ' PriceCalculationResponseDetailed:http://
schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts ' is not
```

expected. Consider using a DataContractResolver or add any types not known statically to the list of known types — for example, by using the KnownTypeAttribute attribute or by adding them to the list of known types passed to DataContractSerializer.'

(Возникла ошибка при попытке сериализации параметров http://wrox/CarRentalService/2009/10: CalculatePriceResult. Сообщение InnerException было "Тип" Wrox.CarRentalService.Contracts.PriceCalculationResponseDetailed" с названием соглашения данных "PriceCalculationResponseDetailed: http://schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts ' не ожидается. Попробуйте использовать DataContractResolver или добавьте любые типы, не объявленные статически в списке известных типов, например, с помощью атрибута KnownTypeAttribute, или добавьте их в список известных типов, передаваемых в DataContractSerializer.)

Как показывает сообщение об ошибке, эта проблема может быть решена с помощью атрибута KnownType. Атрибут KnownType позволяет наследовать тип данных, который будет добавлен в соглашение о данных, тем самым содействуя правильной сериализации и десериализации.

Нужно использовать либо атрибут [KnownType] в классе данных, либо атрибут [ServiceKnownType] в определении соглашения о службе или соглашения об операции (независимо от того, хотите ли вы везде использовать унаследованный класс или базовый или разрешить эти типы только для специальных контрактов службы или служебных операций).

Простое использование атрибута [KnownType] существует в перечислении всех производных классов в соглашении о данных для базового класса.

Листинг 2.22 демонстрирует использование атрибута [KnownType].

Листинг 2.22. Атрибут KnownType в DataContract


```
[DataContract]
[KnownType(typeof(PriceCalculationResponseDetailed))]
public class PriceCalculationResponse
{
   [DataMember]
   public double Price { get; set; }
}
```

Этот атрибут сообщает DataContractSerializer, что в соглашение о данных в дополнение к базовому соглашению надо включить унаследованный класс. Иерархия наследования сохраняется и в XSD, как видно из листинга 2.23.

Листинг 2.23. XSD и наследование


```
<xs:complexType name="PriceCalculationResponseDetailed">
  <xs:extension base="tns:PriceCalculationResponse">
   <xs:sequence>
    <xs:element minOccurs="0" name="Currency" type="xs:string" />
   </xs:sequence>
```

Если экземпляр наследуемого класса будет встречаться только в некоторых методах, или если унаследованные версии допускаются во всех операциях соглашения о службе, атрибут [ServiceKnownType] можно использовать либо на уровне службы

(листинг 2.24), либо на уровне операций, как показано в листинге 2.25. Использование [ServiceKnownType] на уровне операций означает, что унаследованный класс является допустимым только для этого метода. Использование атрибута на уровне службы означает, что унаследованные версии являются допустимыми для всех методов.

Листинг 2.24. Атрибут [ServiceKnownType] для одного метода


```
[OperationContract]
[ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
PriceCalculationResponse CalculatePrice(PriceCalculationRequest request);
```

Листинг 2.25. Атрибут [ServiceKnownType] на уровне службы


```
[OperationContract]
[ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
PriceCalculationResponse CalculatePrice(PriceCalculationRequest request);
```

Если загрузить эту информацию в файл подкачки в раздел System.Runtime.Serialization, получится более гибкий вариант KnownType, как показано в листинге 2.26.

Листинг 2.26. Раздел DataContractSerializer в файле конфигурации


```
<system.runtime.serialization>
    <dataContractSerializer>
      <declaredTypes>
        <add type="Wrox.CarRentalService.
           Contracts.PriceCalculationResponse,
           Wrox.CarRentalService.Contracts">
          <knownType type="Wrox.CarRentalService.Contracts.
           PriceCalculationResponseDetailed,
           Wrox.CarRentalService.Contracts"/>
        </add>
      </declaredTypes>
    </dataContractSerializer>
  </system.runtime.serialization>
```

Существует некоторая негибкость в указании предыдущим типам (KnownType, ServiceKnownType, config) WCF, какие унаследованные классы поддерживаются. Например, каждый раз, когда получается новый производный класс, приходится добавлять дополнительный атрибут [KnownType] к основному соглашению. Таким образом, WCF предлагает вам другие возможности для нахождения KnownTypes. Можно не указывать фиксированное множество производных классов, а резервировать имя метода, который возвращает список производных классов. Это может быть сделано как с помощью атрибутов [KnownType], так и с помощью атрибутов [ServiceType]. Листинг 2.27 демонстрирует использование атрибута [KnownType] со статическим методом GetTypes.

Листинг 2.27. Динамические известные типы


```
[DataContract]
[KnownType("GetTypes")]
public class PriceCalculationResponse
```

```
{
   static Type[] GetTypes()
   {
     Type[] t = { typeof(PriceCalculationResponseDetailed) };
     return t;
   }
}
```

Управление версиями соглашений о службе и данных

Разработка программного обеспечения в большой степени определяется текущими изменениями в требованиях и, следовательно, в предложенной функциональности и структуре данных в сообщениях. В принципе, любое незначительное изменение в операциях или данных может привести к изменению в документе WSDL, что приводит к новой версии службы. Однако, вообще говоря, ни организационно, ни технически невозможно постоянно обновлять компоненты службы и реагировать на изменения в предлагаемых службах.

Поэтому в этой главе будут продемонстрированы методы и приемы, которые позволят поддерживать обмен совместимыми сообщениями, несмотря на изменения в соглашениях о службе или данных. Эти изменения также рассматриваются как изменения, которые не приводят к отмене соглашения. Хотя соглашения не должны быть идентичны, важно, чтобы они были согласованы. В случае службы это означает, например, что добавление нового метода не будет приводить к несогласованности, так как клиент не знаком с этим новым методом вообще. То же самое относится к соглашению о данных; если, например, добавляется новое дополнительное поле, потому что `DataContractSerializer` просто игнорируют это поле и заполняет его значениями по умолчанию.

Однако в определенных ситуациях не избежать принятия нового соглашения: например, если изменилось имя операции, или если добавляется обязательное поле. Изменения такого рода приводят к изменению контракта, и тогда необходимо создать новую версию. В этом случае лучше всего адаптировать пространство имен к новой версии и разместить службу на новой конечной точке.

Управление версиями соглашения о данных

И WCF, и `DataContractSerializer` очень терпимы к изменениям в структуре соглашения о данных.

Например, легко добавить дополнительные элементы данных или исключить элементы данных, в которых атрибут `IsRequired` установлен равным `false`. Одним из преимуществ этого является то, что нет необходимости беспокоиться о таких мелких деталях, однако, в среде, ориентированной на службу, важно придерживаться соглашений или, если изменения необходимы, объявлять их формально и официально.

Предположим, что клиент вносит следующие изменения в версии 1 соглашения о службе и данных, показанного в листинге 2.28.

Листинг 2.28. Версия 1 соглашения о службе и данных


```
[ServiceContract(
    Namespace = "http://wrox/CarRentalService/2009/10",
    Name = "RentalService")]
```

```
    public interface ICarRentalService
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
          PriceCalculationRequest  request, int priority
        );
    }

[DataContract]
public class PriceCalculationRequest
    {
        [DataMember]
        public DateTime PickupDate { get; set; }
        [DataMember]
        public DateTime ReturnDate { get; set; }
        [DataMember]
        public string PickupLocation { get; set; }
        [DataMember]
        public string ReturnLocation { get; set; }

    }
```

Изменения в операциях и соглашениях о данных, показанные в листинге 2.29, восстанавливают согласованность соглашений.

Листинг 2.29. Версия 2 соглашения о службе и данных


```
[ServiceContract(
    Namespace = "http://wrox/CarRentalService/2009/10",
    Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResponse CalculatePrice
   (
     PriceCalculationRequest  request, string someData
   );
}

[DataContract]
public class PriceCalculationRequest
{
   [DataMember]
   public DateTime PickupDate { get; set; }
   [DataMember]
   public DateTime ReturnDate { get; set; }
   [DataMember]
   public string PickupLocation { get; set; }
   [DataMember]
   public string Color { get; set; }
}
```

Из предыдущего примера можно сделать следующие выводы.

- Параметры метода могут быть удалены или добавлены.
- Элементы данных могут быть удалены или добавлены.

- Типы данных могут быть изменены, если при этом сохранена их совместимость.
- Методы могут быть добавлены.

Эти изменения не приведут к каким-либо проблемам технического характера. Однако необходимо, например, определить детали процедуры обработки пропущенных значений. Вполне возможно, что какое-нибудь конкретное значение больше не понадобится и поэтому может быть проигнорировано без каких-либо проблем.

Управление версиями соглашений о данных "туда и обратно"

Проблема, которая возникает снова и снова в связи с версиями соглашений о данных, состоит в добавлении атрибута, десериализация которого в старом соглашении о данных игнорируется.

Давайте более подробно рассмотрим пример, относящийся к запросу цены, — клиент посылает службе запрос о цене и получает ответ в объекте, содержащем признак и цену. Клиент может подтвердить предложенную цену, снова вызвав метод `ConfirmPrice`, как показано в листинге 2.30.

Листинг 2.30. Соглашение о службе `ConfirmPrice`


```
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10
   Name = "RentalService")]
public interface ICarRentalService
{
[OperationContract]
PriceCalculationResponse CalculatePrice
   (
     PriceCalculationRequest request
   );
[OperationContract]
bool ConfirmPrice(PriceCalculationResponse resp);
}
[DataContract]
public class PriceCalculationResponse
{
  [DataMember]
  public int Flag { get; set; }
  [DataMember]
  public string Price { get; set; }
}
```

Листинг 2.31. Код клиента `ConfirmPrice`


```
using (RentalServiceClient carRentalClient = new RentalServiceClient())
{
   PriceCalculationRequest req = new PriceCalculationRequest();
   req.PickupDate = System.DateTime.Now.AddDays(5);
   req.ReturnDate = System.DateTime.Now.AddDays(7);
   req.PickupLocation = "Graz";
   req.ReturnLocation = "Villach";
   PriceCalculationResponse resp;
```

```
   resp = carRentalClient.CalculatePrice(req);
   Console.WriteLine("Price {0}", resp.Price);
   Console.WriteLine(carRentalClient.ConfirmPrice(resp));
}
```

При необходимости интернационализации нужно в соглашение о данных добавить атрибут `Currency`.

Ваш служба все еще может использовать код клиента, поскольку дополнительное поле не приводит к каким-либо проблемам, если свойство `IsRequired` не применяется. Однако значение атрибута `Currency` теряется на стороне клиента, поскольку клиент не знаком с этим недавно добавленным атрибутом и соответственно не может десериализировать его содержание.

Интерфейс `IExtensibleDataObject`

Решение проблемы кругового обращения заключается в реализации интерфейса `IExtensibleDataObject` и присвоении статуса, необходимого полю `ExtensionDataObject`. Если `DataContractSerializer` обнаруживает неизвестные элементы в документе XML, они записываются в свойство `ExtensionDataObject` во время десериализации. Содержание `ExtensionDataObject` сохраняется и при дальнейшем использовании этого объекта, в результате чего не происходит потери данных при смене различных версий соглашений о данных.

Если вы используете диалоговое окно Add Service Reference (Добавить ссылку на службу), классы данных на стороне клиента автоматически реализовывают этот интерфейс. Если хотите выполнить ответные действия на стороне сервера, необходимо реализовать этот интерфейс вручную в классах данных; код в листинге 2.32 показывает, как это сделать.

Листинг 2.32. Интерфейс `IExtensibleDataObject`

Доступно для загрузки на Wrox.com

```
[DataContract]
public class PriceCalculationResponse :IExtensibleDataObject
{
   public ExtensionDataObject ExtensionData { get; set; }

   [DataMember]
   public int Flag { get; set; }
   [DataMember]
   public double Price { get; set; }
   [DataMember]
   public string Currency { get; set; }
}
```

Вы также можете защитить `ExtensionData` в целом, установив в файле конфигурации свойство `<DataContractSerializer ignoreExtensionDataObject = "True" />`. Необходимо соблюдать особую осторожность при работе с расширением данных на стороне сервера, поскольку это может пробить брешь в безопасности.

Рекомендации по управлению версиями соглашения о службе

Если требуется точное соответствие между схемами, необходимо присвоить новую версию вашему соглашению при каждой новой реализации изменений.

Если нет необходимости точного соответствия между схемами, вы должны учесть следующие моменты.

- В любое время можно добавить новые методы.
- Можно удалять любые существующие методы.
- Типы данных параметров должны оставаться совместимыми.

Рекомендации по управлению версиями соглашения о данных

Если требуется точное согласование схем, каждый раз, когда вы выполняете реализацию изменений, вы должны присвоить вашему соглашению новую версию.

Если нет необходимости в точном согласовании схем, нужно учесть следующие моменты.

- Соглашениям о данных не обязательно присваивать новые версии из-за наследования. Вместо этого создайте новый независимый класс данных.
- Для облегчения кругового обмена данными следует в самом начале реализовать интерфейс `IExtensibleDataObject`.
- Если необходимо изменить имя класса данных или некоторых его членов, с помощью атрибутов `DataContract` и `DataMember` можно создать совместимые соглашения о данных.
- Нельзя вносить последующие изменения в типы данных.
- Нельзя изменять порядок появления членов данных с помощью свойства `[DataMember (Order=?)]`.
- Нельзя изменять присвоенное по умолчанию значение `IsRequired` (`false`).
- В любое время можно добавить дополнительные элементы данных, но это изменит порядок сериализации. Чтобы избежать этой проблемы, можно установить свойство `Order` для новых членов равным значению в текущей версии. Таким образом, данным-членам, добавленным в версии 2, должно быть присвоено значение `Order = 2`.
- Данные-члены не должны удаляться.
- Свойство `IsRequired` не должно подвергаться последующим изменениям.

Листинг 2.33 демонстрирует некоторые рекомендации.

Листинг 2.33. Оптимальные способы управления версиями

Доступно для загрузки на Wrox.com

```
[DataContract(
   Name="PriceCalculationRequest",
   Namespace="http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts")]
public class PriceReq //Version 1
{
   [DataMember()]
   private DateTime PickupDate { get; set; }
   [DataMember()]
   public DateTime ReturnDate { get; set; }
   [DataMember()]
   public string PickupLocation { get; set; }
   [DataMember()]
   public string ReturnLocation { get; set; }
```

```
}

[DataContract(
   Name = "PriceCalculationRequest",
   Namespace = "http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts")]
public class PriceReq // Version 2
   [DataMember(Name = "PickupDate")]
   private DateTime FromDate { get; set; }
   [DataMember(Name = "ReturnDate")]
   public DateTime ToDate { get; set; }
   [DataMember(Order = 2)]
   public string PickupLocation { get; set; }
   [DataMember()]
   public string ReturnLocation { get; set; }
   [DataMember(Order=2)] // из Version 2
   public string CarType { get; set; }
}
```

Соглашения о сообщениях

В начале этой главы было разъяснено, что сообщения SOAP передаются между клиентом и сервером. Однако до сих пор вы могли только влиять на содержимое тела сообщения SOAP. В большинстве же случаев рекомендуется управлять соглашениями о данных и операциях. Но в некоторых ситуациях вы можете получить полный контроль над всем сообщением SOAP (заголовок и тело). Заголовок SOAP является подходящим методом для передачи дополнительных данных без расширения класса данных (например, признак защиты).

Соглашения о сообщениях позволяют полностью управлять содержанием заголовка SOAP, так же как и структурой тела сообщения SOAP. До сих пор соглашения о данных использовались как передаваемые или возвращаемые параметры, но теперь они просто заменяются соглашениями о сообщениях. Если вы предпочтете соглашения о сообщениях, важно использовать соглашения о сообщениях для всех параметров, так как методы не могут содержать смесь соглашений о данных и сообщениях.

Листинг 2.34 показывает, как используются три атрибута: `[MessageContract]`, `[MessageHeader]` и `[MessageBody]`. В дополнение к объекту запроса цены в заголовке SOAP также передается имя пользователя.

Листинг 2.34. Атрибуты `[MessageContract]`, `[MessageHeader]` и `[MessageBody]`

Доступно для загрузки на Wrox.com

```
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10",
   Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResp  CalculatePrice(PriceCalculation  request);
}

[MessageContract]
public class PriceCalculation
{
```

```
   [MessageHeader]
   public CustomHeader SoapHeader { get; set; }

   [MessageBodyMember]
   public PriceCalculationRequest PriceRequest { get; set; }
}

[DataContract]
public class CustomHeader
{
   [DataMember]
   public string Username { get; set; }
}

[DataContract]
public class PriceCalculationRequest
{
   [DataMember]
   public DateTime PickupDate { get; set; }
   [DataMember]
   public DateTime ReturnDate { get; set; }
   [DataMember]
   public string PickupLocation { get; set; }
   [DataMember]
   public string ReturnLocation { get; set; }
     }
```

В листинге 2.35 приводится использование соглашения о сообщениях на стороне клиента, а в листинге 2.36 приводится сообщение SOAP.

Листинг 2.35. Соглашения о сообщениях на стороне клиента


```
using (RentalServiceClient carRentalClient = new RentalServiceClient())
{
   PriceCalculation calc = new PriceCalculation();
   PriceCalculationRequest req = new PriceCalculationRequest();
   req.PickupDate = System.DateTime.Now.AddDays(5);
   req.ReturnDate = System.DateTime.Now.AddDays(7);
   req.PickupLocation = "Graz";
   req.ReturnLocation = "Villach";
   calc.PriceRequest = req;

   CustomHeader sHeader = new CustomHeader();
   sHeader.Username = "Johann";
   calc.SoapHeader = sHeader;

   PriceCalculationResp resp=  carRentalClient.CalculatePrice(calc);
}
```

Листинг 2.36. SOAP и заголовки сообщений


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <h:SoapHeader xmlns:h="http://wrox/CarRentalService/2009/10"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
```

```
    <Username xmlns="http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts">
    Johann
    </Username>
  </h:SoapHeader>
 </s:Header>
<s:Body>
 <PriceCalculation xmlns="http://wrox/CarRentalService/2009/10">
  <PriceRequest
 xmlns:a=
"http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
 xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <a:PickupDate> 2010-01-15T14:15:08.1683905+01:00 </a:PickupDate>
   <a:PickupLocation> Graz </a:PickupLocation>
   <a:ReturnDate> 2010-01-17T14:15:08.1693671+01:00 </a:ReturnDate>
   <a:ReturnLocation> Villach </a:ReturnLocation>
  </PriceRequest>
 </PriceCalculation>
</s:Body>
</s:Envelope>
```

Другие примеры использования соглашения о сообщениях приведены в главе 7.

XML сериализации

Как уже упоминалось во введении, WCF поддерживает несколько типов сериализации.

По умолчанию WCF использует класс `DataContractSerializer` для сериализации ваших объектов CLR. В предыдущих примерах вы видели также использование различных свойств в связи с атрибутами `[DataContract]` и `[DataMember]`. Кроме того, вы можете управлять структурой сгенерированного набора информации XML.

`DataContractSerializer` оптимизирован по производительности, поэтому он предлагает меньше возможностей, поскольку не учитывает влияния на структуру данных. Если эти возможности не соответствуют вашим потребностям, можно вернуться к сериализатору XML. Сериализатор XML предлагает большую свободу в сценариях совместной работы и взаимодействии со старыми веб-службами ASP.NET.

Чтобы указать, что WCF должен использовать сериализатор XML, можно просто присвоить соглашению о службе атрибут `[XmlSerializerFormat]`. Если же нужно использовать сериализатор XML только для некоторых операций, к ним также можно применить этот атрибут. Заметьте, что сериализатор XML также сериализирует все общие свойства по умолчанию. Более подробно эти классы описаны в стандартизованной документации на MSDN.

В листинге 2.37 показано использование `[XMLSerializerFormat]` с атрибутами из пространства имен `System.XML.Serialization`, а в листинге 2.38 приведен файл XSD для типа `PriceCalculationRequest`.

Листинг 2.37. Сериализация XML


```
[XmlSerializerFormat]
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10",
```

```
   Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResponse CalculatePrice
   (
    PriceCalculationRequest request
   );
}

[XmlRoot("PriceRequest")]
public class PriceCalculationRequest
{
   [XmlAttribute()]
   public string PickupLocation { get; set; }
   [XmlElement(ElementName="ReturnLoc")]
   public string ReturnLocation { get; set; }
   [XmlIgnore()]
   public DateTime PickupDate { get; set; }
   private DateTime ReturnDate { get; set; }
}
```

Листинг 2.38. XSD для типа `PriceCalculationRequest`


```
<xs:element name="CalculatePrice">
 <xs:complexType>
  <xs:sequence>
<xs:element minOccurs="0" maxOccurs="1"
 name="request"
 type="tns:PriceCalculationRequest" />
</xs:sequence>

</xs:complexType>
</xs:element>
<xs:complexType name="PriceCalculationRequest">
 <xs:sequence>
  <xs:element minOccurs="0" maxOccurs="1" name="ReturnLoc"
  type="xs:string" />
  </xs:sequence>
 <xs:attribute name="PickupLocation" type="xs:string" />
</xs:complexType>
<xs:element name="CalculatePriceResponse">
 <xs:complexType>
  <xs:sequence>
   <xs:element minOccurs="0" maxOccurs="1"
 name="CalculatePriceResult"
 type="tns:PriceCalculationResponse" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
.
</xs:complexType>
```

3

Соединения

В ЭТОЙ ГЛАВЕ...

- Понятие соединения
- Определение адресов
- Прикладные характеристики
- Создание собственных соединений

Windows Communication Foundation (WCF) предлагает программную оболочку, позволяющую забыть о сложности создания служб. Это можно сказать практически обо всех элементах служб, и, вероятно, соединение является самой важной из них. Оно позволяет программисту сконцентрироваться на рассматриваемой проблеме и не заботиться о создании архитектуры, позволяющей вашей системе работать. Эта архитектура уже разработана.

Для описания конечной точки службы (см. предыдущие главы) необходимо реализовать основные элементы WCF: адрес, соединение, соглашение или где, как и что делает служба.

- Адрес представляет собой местонахождение службы в виде ссылки либо в виде информации для клиента о том, куда посылать сообщения.
- Соединение определяет, как работает служба, как отправляются и принимаются сообщения.
- Соглашение определяет содержимое сообщения. (О типах соглашений см. в главе 2.)

Вот эта тройка как раз и представляет собой конечную точку WCF, которая должна быть указана как для клиента, так и для службы.

Соединение состоит из трех основных частей.

- Транспортный протокол. Протокол, который используется для передачи данных. Например, Http или TCP.
- Формат кодирования. Детали кодирования сообщения для передачи по сети, т.е. является ли оно простым текстом, двоичными данными и т.п.
- Другие детали протокола сообщения. Используются для коммуникационного канала. Это может быть требование надежности или безопасности, как в сообщении SOAP.

Соединение представляет собой группу связанных элементов; каждый элемент отвечает за различные детали, такие как кодирование или безопасность сообщения (полный список связывающих элементов WCF приведен ниже в главе). Заранее сконфигурированное соединение WCF объединяет группу связанных элементов для указания на общий сценарий, необходимый разработчику и обобщающий сценарии соединения определенных образцов. Поскольку каждый канал в соединении является частью общей системы коммуникации, это позволяет повторно использовать и создавать собственные соединения с использованием уже существующих и заранее сконфигурированных элементов. Вы можете использовать одно из множества предопределенных соединений, которые предлагает оболочка WCF. Такие соединения представляют собой наборы наиболее часто используемых различных сценариев. Если соединения, предложенные оболочкой, вас не устраивают, можете выбрать один из вариантов, предложенных ниже.

Сначала можете попытаться настроить наиболее подходящее из предопределенных соединений, устанавливая некоторые из его свойств. Также можно создать собственное соединение, выбирая существующие элементы и объединяя их в группы.

В большинстве случаев подойдет одно из предопределенных WCF соединений. Это невероятно уменьшает время разработки, так как не нужно писать код для передачи сообщений. Вы можете определить только основные элементы, — все остальное уже сделано. Поскольку WCF уже реализует соединения, можете получить несколько конечных точек с различными соединениями. Это и есть важнейшая часть WCF — функциональная совместимость. Благодаря ей одни и те же методы службы могут реализовывать несколько различных протоколов и видов передачи сообщений. Это позволяет быстро предложить ваши службы различным типам клиентов, которые в них нуждаются. Вы можете использовать специфические соединения .NET для общения с клиентами .NET и использовать преимущества безопасности и производительности, встроенной в WCF. Используя стандарты WS-*, вы также можете предложить вашу службу клиентам на Java, PHP и любым другим, соответствующим спецификации. С другой стороны, можно использовать соединение, которое позволяет веб-клиентам подключаться к вашей службе с помощью упрощенного интерфейса, известного как интерфейс REST (Representational State Transfer). Это также позволяет быстро подкачивать соединение или изменить соединение для проверки производительности.

Что касается SOA, то REST предоставляет альтернативный способ выполнения операций на дистанционном ресурсе. Работа с REST рассматривается более подробно в главе 4.

При выборе лучшего соединения для вашей системы можете создать множество конечных точек с различным порядком и вариантами конфигурации, т.е. можно запустить тесты производительности (измерение времени отклика) для каждого соединения.

Логика и соглашение, а также бизнес-логика не изменяются, меняется только конфигурация соединения. Это означает, что просто изменяется способ передачи. Выполнение этого без WCF предполагает изучение дистанционной работы с .NET, ASMX веб-служб и всего, что с ними связано. Затем необходимо реализовать каждый из них и разместить их все различными способами перед тестированием. Если необходимо разместить реализации на вашем рабочем сервере, то опять же их нужно разместить в различных местах в зависимости от технологии. Здесь нетрудно увидеть преимущества WCF для всех сценариев SOA.

Как работают соединения

Как описано выше, WCF позволяет передавать и принимать сообщения с использованием различных транспортных протоколов. Для установления связи вашей службы с внешним миром можно также выбрать кодировку сообщения (текст, данные в двоичном формате или Mtom (SOAP Message Transmission Optimization Mechanism)) или стандартный протокол для легко управляемого, безопасного и надежного взаимодействия. Соединение является ключевым элементом архитектуры WCF и позволяет описывать, как сообщение обрабатывается перед специфической для службы реализацией на стороне сервера или перед реализацией для клиента на стороне потребителя.

Соединение состоит из набора компонентов. Как это происходит, поясняется ниже в главе. Пока достаточно знать, что каждый элемент соединения соответствует каналу передачи и протоколу, расположенным в некоем хранилище, называемом *стеком канала*. Стек канала представляет собой последовательность каналов, через которую проходит сообщение до получения исполняемой программой.

Тип и порядок элементов связывания важны, так как они определяют использование каналов в процессе выполнения службы (рис. 3.1).

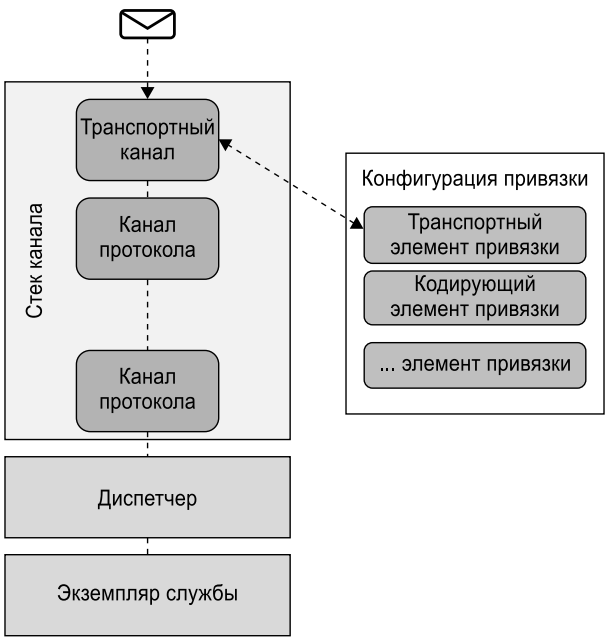


Рис. 3.1. Тип и порядок элементов

Заметьте, что транспортный и кодирующий элементы должны быть указаны первым и вторым, поскольку они могут корректировать взаимодействия каналов. На самом деле поступающее сообщение принимается транспортным каналом (т.е. каналом TCP, который прослушивает предопределенный порт) и передается кодирующему каналу. Наконец, оно поступает во все каналы протокола, определенные в соединении через его элементы.

Также возможна реализация пользовательских каналов для получения других сценариев, но необходимость создания вашего собственного способа передачи вместо HTTP- или TCP-протоколов встречается редко. Однако вы должны знать, что можете это сделать и что WCF поддерживает такой уровень гибкости.

Все эти подробности рассматриваются в данной главе, но прежде всего необходимо прояснить концепцию — что такое адресация и режимы и как они могут использоваться в архитектуре WCF.

Адресация

Адрес является первым основным элементом WCF. Каждая конечная точка должна иметь свой уникальный адрес, который является местом расположения службы. Этот адрес может быть IP-адресом, именем сервера, URL и т.п. Адрес нужен для нахождения службы и подобен URL для веб-сайта. Он содержит много информации, необходимой для подключения к службе, в том числе и непосредственно в адресе. Адрес состоит из нескольких частей.

- Транспортная схема. Начальная часть адреса, которая содержит протокол, используемый службой.
- Расположение службы. Действительный адрес, по которому находится служба. Он может быть локальным в вашей сети (`http://localserver/`) или веб-адресом (`http://www.webserver.com` или `http://216.239.59.104`).
- Порт. Может быть указан, если для данного протокола не используется порт по умолчанию.
- Путь. Относительное место расположения ресурса на сервере. Если имя файла ресурса не указано, сервер может возвратить файл по умолчанию.

Формат адреса может быть таким.

```
scheme://SERVERLOCATION[:port]/path/subpath
```

Вот простой пример.

```
http://myserver.com/service
```

Из этого адреса видно, что транспортной схемой является протокол Http, адресом сервера является `myserver.com`, порт не определен, что означает использование порта Http (а именно 80) и путь `/service`. В табл. 3.1 приведены примеры для каждого транспортного протокола.

Таблица 3.1. Примеры WCF адресов для различных транспортных протоколов

Транспортный протокол	Пример адреса
Http	`http://localhost:8001` `http://localhost:8001/Service1`
Http (защищенный)	`https://localhost:8001`

Окончание табл. 3.1

Транспортный протокол	Пример адреса
TCP Peer network	`net.tcp://localhost:8002/Service1` `net.p2p://localhost/`
IPC (Inter-process communication over named pipes)	`net.pipe://localhost/PipeService1`
MSMQ (Microsoft Message Queue)	`net.msmq://localhost`

Транспортные протоколы

WCF позволяет взаимодействовать с клиентами с помощью любого протокола на ваш выбор. В отличие от веб-служб ASMX, работающих исключительно через HTTP, теперь вы можете выбрать протокол с учетом вашего клиента.

- **http**. Предпочтительный протокол для общения через веб. Позволяет интегрировать службу с открытыми стандартами и обслуживать клиентов с различными архитектурами.
- **TCP**. Быстрый протокол двоичного формата. Позволяет осуществлять высокопроизводительное взаимодействие WCF — WCF и является лучшим для интранет-сценариев.
- **Именованные каналы.** Позволяют установить быстрое и надежное сообщение между клиентом и службой при запуске на одном компьютере. Работают только в случае WCF — WCF и используют общий участок памяти для различных процессов.
- **MSMQ.** Протокол Microsoft Message Queue позволяет организовать очереди и очень полезен при разъединенном взаимодействии клиента и службы. Транспорт net.msmq применяется тогда, когда клиент хочет поставить в очередь сообщение, которое служба может обработать позже.
- **Пользовательский протокол.** WCF разрешает общение по сети с помощью всех перечисленных протоколов. Хотя очень редко, но все же может потребоваться создание собственного протокола.

Режимы работы

Применяя различные режимы работы к различным частям системы, можно влиять на службы WCF, т.е. управлять сессиями, параллелизмом, регулировать нагрузки и транзакции. Некоторые из этих возможностей могут применяться только на определенных уровнях: служба, конечная точка, операция и соглашение.

В зависимости от уровня существует несколько способов определения режима работы. Многое может быть сделано через конфигурацию или атрибуты. Но что-то должно быть реализовано непосредственно в коде, хотя многое может быть записано в конфигурации.

Режимы работы применяются локально к клиенту или службе, поэтому они не выводятся в службы WSDL.

Режимы работы службы

Атрибут `[ServiceBehavior]` позволяет применять правила и режимы ко всей службе. Во время разработки можно управлять поведением параллельных вычислений, экземпляров, регулировки, транзакций, сессий и потоков путем назначения их свойств.

- `AddressFilterMode`. Позволяет изменить фильтр сообщений. Свойство `AddressFilterMode` имеет три значения: `Any`, `Exact` и `Prefix`. Эти установки позволяют диспетчеру определить правильную конечною точку, ответственную за обработку входного сообщения.
- `AutomaticSessionShutdown`. Это булево поле, которое предотвращает остановку сессии сервером, когда обработаны все сообщения. По умолчанию оно установлено равным `true`; установив его равным `false`, можно регулировать продолжительность сессии.
- `ConcurrencyMode`. Указывает, может ли служба работать в одном потоке или во множестве потоков. По определению оно установлено так, что допускает только один поток (`Single`). Установка для *нескольких потоков* (multiple) означает, что в вашей службе необходимо реализовать безопасность потока.
- `IgnoreExtensionDataObject`. Булево поле, `false` по умолчанию. Если установить его равным `true`, то любые дополнительные неизвестные сериализуемые данные не будут посылаться в сообщении.
- `IncludeExceptionDetailInFaults`. Это свойство необходимо установить равным `true`, если хотите отправить необрабатываемое сообщение об ошибке клиенту в качестве сбоя SOAP. В этой ситуации генерируется большое сообщение SOAP с некоторой внутренней информацией (т.е. трассировкой стека), которую обычно клиент знать не должен. Установите это свойство равным `true` в среде разработки, но в рабочей среде установите для него значение `false`.
- `InstanceContextMode`. Это свойство используется для определения времени существования экземпляра службы. Допустимые значения — `PerSession`, `PerCall` и `Single`. Управление экземплярами подробно рассматривается в главе 4.
- `MaxItemsInObjectGraph`. Устанавливает максимальное число элементов в графе сериализации/десериализации объектов. Иногда можно получить исключение, если число элементов в сериализируемом или десериализируемом объекте превысило максимально допустимое. Увеличьте это число до необходимого для вас.
- `ReleaseServiceInstanceOnTransactionComplete`. Если это свойство имеет значение `true`, объект службы разрушается по окончании активной транзакции.
- `TransactionAutoCompleteOnSessionClose`. Присвойте этому свойству значение `true`, если хотите отметить активную транзакцию как законченную по окончании сессии без ошибок.
- `TransactionIsolationLevel`. Укажите доступный уровень изоляции для текущего объекта, когда транзакция активна. Возможные значения — `Serializable`, `RepeatableRead`, `ReadCommitted`, `ReadUncommitted`, `Snapshot`, `Chaos` и `Unspecified`.
- `TransactionTimeout`. Иногда транзакции могут выполняться долго. Вы можете задать тайм-аут, после которого транзакция считается прерванной и запускается процесс отката.
- `UseSynchronizationContext`. Это свойство может быть использовано для указания на необходимость совместимости между потоками пользовательского интерфейса и службы.
- `ValidateMustUnderstand`. Используется для проверки правильности заголовков SOAP, которые отмечены как `MustUnderstand`. Это булево поле, и по умолчанию оно устанавливается равным `false`.

Для использования атрибута [ServiceBehavior], как и в случае всех остальных атрибутов в .NET, вы должны выделить класс службы и назначить соответствующе свойства (листинг 3.1).

Листинг 3.1. Использование режима работы службы


```
using System;
using Wrox.CarRentalService.Contracts;

namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
    public class CarRentalService : ICarRentalService
    {
        // действия службы
    }
}
```

В данном примере [ServiceBehaviorAttribute] используется для установки времени существования экземпляра службы в режим PerCall.[ServiceBehaviorAttribute] является всего лишь одним из специальных режимов службы. Напротив, ServiceBehavior реализует специфический интерфейс, называемый IServiceBehavior, со следующими методами.

- AddBindingParameters. Этот метод, вызываемый только один раз для каждой конечной точки, используется для проверки описания текущей службы и модификации параметров соединения в соответствии с требованиями службы.
- ApplyDispatchBehavior. Это самый важный метод в интерфейсе. Можете использовать его для применения режима работы службы в исполняемой программе. В качестве примера можно привести установку MessageInspector для регистрации входящих и исходящих сообщений.
- Validate. Как следует из названия, этот метод должен применяться для проверки описания службы и возможной передачи исключений, если оно не соответствует требованиям режима службы.

Кроме ServiceBehaviorAttribute, существует множество других режимов, реализующих интерфейс IServiceBehavior, конфигурируемый как с помощью файла конфигурации, так и программно. Одним из наиболее часто используемых режимов является ServiceMetadataBehavior, который позволяет публиковать метаданные службы.

Как показано в листинге 3.2, в файле конфигурации, в секции <system.serviceModel> <behaviors>, можно указать желательный режим службы.

Листинг 3.2. Разрешение публикации метаданных службы в файле


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.CarRentalService.Implementations.Europe.CarRentalService"
behaviorConfiguration="CarRentalServiceBehavior">
        <endpoint address="http://localhost:8000/CarRentalService"
        contract="Wrox.CarRentalService.Contracts.ICarRentalService"
```

```
        binding="basicHttpBinding">
        </endpoint>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="CarRentalServiceBehavior">
          <serviceMetadata httpGetEnabled ="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Приведенная в примере конфигурация разрешает публикацию метаданных службы с помощью `ServiceMetadataBehavior`. Это позволяет клиенту делать запрос и получать информацию, описывающую службу в стандартном формате на основе стандартной спецификации WSDL (Web Services Description Language) или WS-Metadata Exchange. В качестве альтернативы можно использовать программный подход для определения режима службы, как в листинге 3.3.

Листинг 3.3. Разрешение публикации метаданных службы в коде

Доступно для загрузки на Wrox.com

```
using Wrox.CarRentalService.Implementations.Europe;
using Wrox.CarRentalService.Contracts;
...

ServiceHost host = new ServiceHost(typeof(CarRentalService));

ServiceMetadataBehavior serviceMetadata =
host.Description.Behaviors.Find <ServiceMetadataBehavior>  ();
if (serviceMetadata == null)
{
    serviceMetadata = new ServiceMetadataBehavior();
    host.Description.Behaviors.Add(serviceMetadata);
}
serviceMetadata.HttpGetEnabled = true;

BasicHttpBinding binding = new BasicHttpBinding();
host.AddServiceEndpoint(typeof(ICarRentalService),
                        binding,
                        "http://localhost:8080/CarRentalService");
```

В приведенном коде проверяется, имеет ли текущий экземпляр `ServiceHost` экземпляр `ServiceMetadataBehavior`. Если нет, создается новый экземпляр, затем свойство `HttpGetEnabled` устанавливается равным `true`, как в файле конфигурации. Только один экземпляр того же типа `ServiceBehavior` может существовать в каждый момент для каждого `ServiceDescription`.

`ServiceHost` подробно рассматривается в главе 14. Сейчас важно знать только то, что этот параметр используется для размещения и управления экземплярами службы в приложениях .NET, таких как службы Windows, формы Windows и консольные приложения.

Режимы операций

Атрибут `OperationBehavior` позволяет управлять методами класса и конкретизировать определенные области вашей службы. Такими областями являются транзакции, идентификация того, кто вызвал службу, и повторное использование объектов.

- `AutoDisposeParameters`. Этот параметр устанавливает автоматическое разрушение входных, выходных и ссылочных параметров. Значение по умолчанию — `true`.
- `TransactionAutoComplete`. Когда транзакция разрешена, это свойство маркирует транзакцию как законченную, если не произошла ошибка в текущем методе. В противном случае транзакция прерывается. Если это свойство установлено равным `false`, вы должны вручную определять транзакцию как законченную или прерванную.
- `TransactionScopeRequired`. Это свойство определяет, что транзакция является необходимой для текущего метода.
- `Impersonation`. Иногда бывает необходимо выполнить операцию иначе, в зависимости от того, кто вызвал операции. Установите это свойство равным `Required` или `Allowed`, как по умолчанию.
- `ReleaseInstanceMode`. Это свойство позволяет изменять значение `InstanceContextMode` и воспроизводить установки для объекта службы. Возможные значения: `None`, `BeforeCall`, `AfterCall` и `BeforeAndAfterCall`.

`OperationBehaviorAttribute` позволяет управлять свойствами, тесно связанными с операциями службы. Можете просто применить данный атрибут к операции, которой хотите управлять, а затем установить соответствующие свойства, чтобы получить необходимый режим (листинг 3.4).

Листинг 3.4. Определение `OperationBehavior` в реализации службы


```
using System;
using Wrox.CarRentalService.Contracts;

namespace Wrox.CarRentalService.Implementations.Europe
{
    public class CarRentalService : ICarRentalService
    {
        [OperationBehavior(
            AutoDisposeParameters = true,
            Impersonation = ImpersonationOption.NotAllowed,
            ReleaseInstanceMode = ReleaseInstanceMode.None,
            TransactionAutoComplete = true,
            TransactionScopeRequired = false)]
        double CalculatePrice(DateTime pickupDate, DateTime returnDate,
   string pickupLocation, string vehiclePreference)
        {
            // здесь код метода
        }
    }
}
```

В данном примере для всех свойств `OperationBehaviorAttribute` установлены значения по умолчанию.

Режимы конечных точек

Хотя режимы службы применимы только на служебной стороне, режимы конечных точек могут применяться как на серверной, так и на клиентской стороне. Некоторые подробности, такие как использование учетных данных клиента или настройки сериализатора, могут управляться с помощью специфических режимов конечных точек. WCF предлагает предопределенный набор режимов, каждый из которых реализует интерфейс `IEndpointBehavior`.

- `AddBindingParameters`. Используйте этот метод, вызываемый только один раз для каждой конечной точки, для проверки описания службы и изменения параметров соединения в соответствии с требуемым поведением конечной точки.
- `ApplyDispatchBehavior`. Самый важный метод в интерфейсе. С его помощью можете применить логику поведения конечной точки на стороне службы.
- `ApplyClientBehavior`. На стороне клиента можно использовать этот метод для применения логики поведения конечной точки во время выполнения.
- `Validate`. Как видно из названия, этот метод должен быть использован для проверки описания службы и при передаче исключения в случае необходимости, если описание не соответствует требуемому режиму работы конечной точки.

Установить различные режимы поведения конечной точки можно в файле конфигурации, в программном коде, или комбинируя оба эти метода. В отличие от `ServiceBehaviorAttribute` режим конечной точки может применяться как в клиенте, так и в службе. Другое важное отличие на стороне службы состоит в том, что установки режима конечной точки достоверны только на уровне конечной точки, тогда как установки режима службы достоверны на уровне службы. Кстати, некоторая специфическая реализация может разрешить только использование клиентом, как и в режиме `ClientCredential`.

Используя файл конфигурации, можно установить и настроить режимы конечной точки. Например, на стороне клиента можно указать учетные данные, которые клиент должен использовать при вызове относительной конечной точки, с помощью режима `ClientCredential` (листинг 3.5).

Листинг 3.5. Установка `ClientCredentials` в файле конфигурации


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="Wrox.CarRentalService.Implementations.Europe.CarRentalService"
        behaviorConfiguration="CarRentalServiceBehavior">
        <endpoint address="http://localhost:8000/CarRentalService"
          contract="Wrox.CarRentalService.Contracts.ICarRentalService"
          binding="basicHttpBinding"
          behaviorConfiguration="carRentalEndpointBehavior">
        </endpoint>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
```

```
        <behavior name="CarRentalServiceBehavior">
          <serviceMetadata httpGetEnabled ="true"/>
        </behavior>
      </serviceBehaviors>
      <endpointBehaviors>
        <behavior name="carRentalEndpointBehavior">
          <clientCredentials>
            <clientCertificate findValue="CN=client_cert"
                               storeLocation="CurrentUser"
                               storeName="My"
                               x509FindType="FindBySubjectDistinguishedName"/>
          </clientCredentials>
        </behavior>
      </endpointBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Если для настройки режима файл конфигурации не подходит, можно настроить режим программно на стороне клиента (листинг 3.6).

Листинг 3.6. Установка `ClientCredentials` на клиенте с помощью программного кода


```
CarRentalServiceClient client = new CarRentalServiceClient();

try
{
    client.ClientCredentials.ClientCertificate.SetCertificate(
        StoreLocation.CurrentUser,
        StoreName.My,
        X509FindType.FindBySubjectDistinguishedName,
        "CN=client_cert");

    // еще код

    client.Close();
}
catch (Exception ex)
{
    client.Abort();
    throw;
}
```

Код меняется, если вы хотите установить режим конечной точки на стороне сервера (листинг 3.7). Как видно, для режима службы, если вы не хотите использовать файл конфигурации, полезным является класс `ServiceHost`.

Листинг 3.7. Установка `ClientCredentials` на службе с помощью кода


```
using Wrox.CarRentalService.Implementations.Europe;
...

ServiceHost host = new ServiceHost(typeof(CarRentalService));
```

```
try
{
    BasicHttpBinding binding = new BasicHttpBinding();
    ServiceEndpoint serviceEndpoint =
        host.AddServiceEndpoint(typeof(ICarRentalService),
        binding,
        "http://localhost:8080/CarRentalService");

    ServiceCredentials credentials = new ServiceCredentials();
    credentials.ServiceCertificate.SetCertificate(
        StoreLocation.CurrentUser,
        StoreName.My,
        X509FindType.FindBySubjectDistinguishedName,
        "CN=service_cert");

    host.Open();

    // Остальной код ...

    host.Close();
}
catch (Exception)
{
    host.Abort();
}
```

Режим `ClientCredentials` не может использоваться на стороне службы. В этом примере вы видите, как установить защиту службы, используя режим `ServiceCredentials`. Настройки безопасности и другие вопросы защиты в WCF рассматриваются в главах 7–9.

Режимы соглашений

Интерфейс `IContractBehavior` может быть реализован для расширения или модификации любых аспектов ваших соглашений. Его можно применять везде, где используются ваши соглашения. Этот интерфейс имеет четыре метода, которые можно реализовать для модификации соглашений. Режимы соглашений нельзя применять в конфигурации, поэтому делать это следует в коде или в атрибутах.

- `AddBindingParameters`. Позволяет добавить пользовательские параметры, полезные для выполнения в заданном режиме работы. Этот метод вызывается один раз для каждой конечной точки.
- `ApplyDispatchBehavior`. Реализуйте этот метод, применяя логику поведения конечной точки на стороне службы.
- `ApplyClientBehavior`. Используйте этот метод для применения логики поведения конечной точки во время выполнения.
- `Validate`. Полезный метод, если во время выполнения вы хотите проверить контекст для работы в данном режиме.

Что касается `IEndpointBehavior`, то `IContractBehavior` может применяться к интерфейсу соглашения на стороне и клиента, и службы. Но, с другой стороны, реализация не может быть добавлена во время выполнения с помощью файла конфигурации.

Она может быть добавлена только во время разработки с помощью атрибутов или программно.

Имеется только одна предопределенная реализация: `DeliveryRequirementsAttribute`. Этот режим действует как контрольный параметр по отношению к функциональным возможностям, загружаемым во время выполнения из файла конфигурации (листинг 3.8).

Листинг 3.8. Использование режима соглашения


```
using System;

namespace Wrox.CarRentalService.Contracts
{
    [DeliveryRequirements(
    QueuedDeliveryRequirements = QueuedDeliveryRequirementsMode.NotAllowed,
    RequireOrderedDelivery = true)]
    public class ICarRentalService
    {
        // действия службы
    }
}
```

В данном примере `DeliveryRequirements` показывает, что это соглашение не разрешает очередь соглашений, а требует упорядоченной доставки сообщений. Можете также установить этот режим программно (листинг 3.9).

Листинг 3.9. Использование программируемого режима соглашения на стороне клиента


```
DeliveryRequirementsAttribute deliveryRequirements =
    new DeliveryRequirementsAttribute();
deliveryRequirements.RequireOrderedDelivery = true;
deliveryRequirements.QueuedDeliveryRequirements =
    QueuedDeliveryRequirementsMode.NotAllowed;

CarRentalServiceClient client = new CarRentalServiceClient();
client.Endpoint.Contract.Behaviors.Add(deliveryRequirements);
```

На стороне сервера вы можете сделать это иначе, с помощью `ServiceHost`, как показано в листинге 3.10.

Листинг 3.10. Использование программируемого режима соглашения на стороне сервера


```
using Wrox.CarRentalService.Implementations.Europe;
...

ServiceHost host = new ServiceHost(typeof(CarRentalService));

BasicHttpBinding binding = new BasicHttpBinding();
ServiceEndpoint serviceEndpoint =
host.AddServiceEndpoint(typeof(ICarRentalService),
```

```
                        binding,
                        "http://localhost:8080/CarRentalService");

DeliveryRequirementsAttribute deliveryRequirements = new
DeliveryRequirementsAttribute();
deliveryRequirements.RequireOrderedDelivery = true;
deliveryRequirements.QueuedDeliveryRequirements =
QueuedDeliveryRequirementsMode.NotAllowed;
serviceEndpoint.Contract.Behaviors.Add(deliveryRequirements);

host.Open();
```

Соединения

WCF предоставляет несколько различных соединений, которые позволяют разрабатывать свою службу без необходимости писать много кода. Шаблоны и команды, используемые снова и снова другими разработчиками, объединяются вместе, что позволяет вам быстро запустить в производство "стандартные" службы. Если у вас есть различные требования, вы все равно можете разработать пользовательское соединение с минимальными усилиями, но должны знать, что получится, и действительно ли это то соединение, которое отвечает вашим потребностям.

Соединения с префиксом "net" означают, что они разработаны с использованием преимуществ технологии .NET и выполняют много операций, увеличивающих производительность. Соединения с префиксом "ws" могут использоваться во всех системах и соответствуют множеству веб-стандартов. Краткое обсуждение имеющихся в настоящее время стандартных соединений приведено в табл. 3.2.

Таблица 3.2. Стандартные соединения в пространстве имен `System.ServiceModel`

Соединение	Описание
`System.ServiceModel`	Использует профиль WS-I Basic Profile 1.1 (`http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html`), который является профилем для старых ASMX веб-служб. Он позволяет создавать и использовать стили ASMX в WCF. В качестве транспортного протокола по умолчанию используется HTTP для передачи и кодирования сообщений в формате текста UTF-8
`webHttpBinding`	Позволяет публиковать службы в виде HTTP-запросов, как в службах на основе REST, выводящих результаты в виде XML или JSON
`wsHttpBinding`	Использует расширенные профили на основе WS-* для создания веб-служб. Это касается многих профилей, таких как WS-Security, WS-Transactions, WS-BusinessActivity и др.
`wsDualHttpBinding`	Использует те же профили, что и `wsHttpBinding`, но для двунаправленных соглашений. Это означает, что как служба, так и клиент может отправлять и получать сообщения
`wsFederationHttpBinding`	Предназначен для расширенных веб-служб на основе WS-*, использующих интегрированную идентификацию
`netTcpBinding`	Используйте это соединение для передачи данных по TCP между двумя машинами в вашей сети. При этом и клиент, и служба должны использовать WCF. Функции, включая безопасность и транзакции, оптимизированы для WCF
`netNamedPipeBinding`	Оптимизировано для немашинной коммуникации

Окончание табл. 3.2

Соединение	Описание
`netPeerTcpBinding`	Соединение также предназначено для коммуникации по протоколу TCP, но использует одноранговую сеть peer-to-peer (P2P). Каждый из узлов ведет себя, как клиент и сервер по отношению к другим узлам. Полагается на систему определения имен для вычисления каждого узла сети по имени. Peer Name Resolution Protocol (Протокол нахождения узла по имени — PNRP) используется и описывается в соединении.
	P2P активно используется в таких совместно используемых ресурсах, как торренты, которые стали очень популярны в последние несколько лет
`netMsmqBinding`	Соединение для асинхронной коммуникации с использованием MSMQ (Microsoft Message Queue)
`msmqIntegrationBinding`	Это соединение разрешает разработчикам WCF взаимодействовать с существующими системами через MSMQ

Привязки `BasicHttpBinding` и `WSHttpBinding`

В сценарии со множеством разнообразных клиентов, которые должны иметь доступ к вашей службе, использование совместимых протоколов является обязательным условием. WCF предоставляет в ваше распоряжение два типа соединений, использующих в качестве транспортного протокола HTTP (HyperText Transfer Protocol). `BasicHttpBinding` и `WSHttpBinding` — два наиболее совместимых соединения, пригодных к использованию.

`BasicHttpBinding` обеспечивает высокий уровень совместимости между WCF и другими оболочками для создания служб, таких как ASP.NET ASMX. В стандартных терминах это соединение является реализацией спецификации WS-I Basic Profile 1.1 и поддерживает SOAP 1.1 в качестве протокола сообщений. Безопасность базируется на основе транспортного протокола (HTTP), который используется для аутентификации и шифрования, или на основе WS-Security 1.0 для безопасности сообщений. Поскольку использование `BasicHttpBinding` полезно только для обратной совместимости с ASP.NET ASMX, так как ASMX ASP.NET не поддерживает WS-Security, и вся полезная информация отправляется в текстовом формате, то только в редких случаях есть основания для его использования.

Хотя он предлагает большой уровень совместимости, есть много сценариев, в которых он непригоден. `WSHttpBinding`, как следует из названия, позволяет использовать различные WS-* спецификации, такие как WS-Security 1.1, WS-Reliable Messaging (для надежной и упорядоченной доставки сообщений), WS-Atomic Transaction, WS-Trust и др. Это позволяет создавать большие и надежные инфраструктуры служб.

Очевидно, что использование этих спецификаций может вызвать передачу большого количества сообщений между клиентом и службами. Несомненно, поддержка стандартов делает `WSHttpBinding` более надежным и безопасным, чем `BasicHttpBinding`. Наконец, если совместимость не требуется, целесообразно выбрать `WSHttpBinding`.

Привязка `NetTcpBinding`

`NetTcpBinding` позволяет использовать протокол TCP с двоичным кодированием сообщений. По умолчанию все обязательные настройки выключены. Если нужно использовать защиту, придется установить ее. Если нужна надежность и упорядоченная доставка с WS-ReliableMessaging или если вы хотите сделать вашу коммуникацию атомарной, вы должны разрешить это.

NetTcpBinding является более быстрым и надежным протоколом по сравнению с соединением по протоколу HTTP. Однако он полезен только с соединением WCF — WCF. Этот сценарий представляет лучшее прикладное решение.

Привязка NetMsmqBinding

BasicHttpBinding, WSHttpBinding и NetTcpBinding являются решениями для связанных сценариев. Иногда нужно отделить провайдера службы и потребителя. Обычно это происходит, когда служба и клиент обрабатывают сообщение в разное время. Клиент, в свою очередь, должен знать точную конечную точку службы, которая получает и перерабатывает сообщение.

NetMsmqBinding позволяет использовать MSMQ для постановки сообщений в очередь и изъятия их из очереди. Типичный сценарий предполагает, что служба получает сообщения от клиента, ставит сообщение в очередь, затем получает сообщение из очереди и обрабатывает его в выходном буфере. Таким образом, можно разделить клиента и службу, если сообщения передаются в очередь, на которую не налагается никаких ограничений.

В .NET Framework вы найдете другое соединение, использующее MSMQ в качестве транспортного протокола, — MsmqIntegrationBinding. Оно отличается от соединения NetMsmqBinding тем, что допускает коммуникацию с существующими приложениями, использующими System.Messaging или COM API. Вы не сможете использовать NetMsmqBinding для чтения сообщений, записанных в MsmqIntegrationBinding, и наоборот, из-за их полностью различного формата сообщений.

Соединения, допускающие контекст

WCF предоставляет специальный набор допускающих контекст соединений для ряда различных стандартных соединений. Допускающие контекст соединения позволяют посылать дополнительные параметры для обмена контекстными данными с использованием HttpCookies или заголовка SOAP. Эти соединения наследуют свое главное соединение и используются так же.

WCF обеспечивает следующими контекстными соединениями.

- BasicHttpContextBinding
- NetTcpContextBinding
- WSHttpContextBinding

Эти соединения позволяют реализовывать устойчивые службы. Устойчивая служба в WCF — это служба, в которой состояние поддерживается на протяжении различных вызовов. Следующее сообщение SOAP демонстрирует использование заголовка SOAP для передачи контекста между клиентом и услугой (листинг 3.11).

Листинг 3.11. Сообщение SOAP, отправляемое, когда соединение допускает передачу контекста

Доступно для загрузки на Wrox.com

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:a="http://www.w3.org/2005/08/addressing">
    <s:Header>
        <a:Action s:mustUnderstand="1">
          http://tempuri.org/IService1/GetData
        </a:Action>
```

```
        <a:MessageID>
          urn:uuid:32186d9f-573f-4252-989f-ad94a469271b
        </a:MessageID>
        <a:ReplyTo>
            <a:Address>
              http://www.w3.org/2005/08/addressing/anonymous
            </a:Address>
        </a:ReplyTo>
          <Context xmlns="http://schemas.microsoft.com/ws/2006/05/context">
          <Property name="instanceId">
              1507fdf6-a27b-40f2-b339-f331baa937f1
          </Property>
      </Context>
      <a:To s:mustUnderstand="1">http://fabio-nb:10101/IService</a:To>
  </s:Header>
  <s:Body>
        <GetData xmlns="http://tempuri.org/">
            <value>1</value>
        </GetData>
    </s:Body>
</s:Envelope>
```

Заголовок SOAP Context имеет дочерний элемент `Context`, который содержит идентификатор экземпляра службы в элементе `property`. Этот идентификатор используется на стороне сервера для получения экземпляра и возобновления устойчивого состояния службы.

Очень много соединений на выбор

Все эти встроенные соединения построены на стандартах, работающих на специфических сценариях, но также будут работать, если вы выберете неудачный или наименее эффективный сценарий. Так как же выбрать, какой из них подходит для конкретной ситуации?

Самым очевидным будет решение для случая, когда нужно взаимодействовать с приложением, отличным от приложения WCF. Если служба будет обмениваться сообщениями с клиентами WCF, можете использовать одно из соединений с префиксом ".net". Имейте в виду, что можно указать несколько конечных точек для одной службы, которые будут использоваться различными соединениями. Это позволяет воспользоваться преимуществами оптимизации производительности для клиентов WCF, но при этом все равно поддерживаются службы ASMX и WS-* стандарты. Таким образом обеспечивается безопасность и другие возможности в интересах многих клиентов.

Например, если хотите установить связи WCF—WCF, выбор `NetTcpBinding` с двоичным кодированием делает общение в четыре или пять раз быстрее, чем `BasicHttpBinding` с текстовым кодированием.

Предлагаемые в WCF соединения должны обеспечивать все, что вам нужно, однако если вам нужно что-то изменить или создать полностью новое соединение, WCF позволяет сделать это очень просто. К данному вопросу мы еще вернемся в этой главе.

Конфигурирование соединений

Можно определить соединение вместе с конечной точкой как в файле конфигурации, так и в коде. Общепринято определять конечную точку в конфигурации или

в коде, поскольку во время разработки вы обычно не знаете, где будет размещаться служба. Даже если вы имеете представление о том, где она будет, определяя конечную точку в конфигурации, можете в дальнейшем изменить ее без пересборки кода и переустановки службы. Служба может располагаться в одном месте в течение многих лет, но ее месторасположение может измениться в связи с изменением инфраструктуры компании или организации. Предлагаемый подход позволяет администраторам внести эти изменения без вас, редактируя XML. Другие разработчики также могут внести эти изменения без копания в коде.

Конфигурация находится в файле `App.config` (или `Web.config`). Вы можете отредактировать его непосредственно (как в предложенном примере) или, используя WCF Configuration Editor (см. рис. 3.2), щелкнуть правой кнопкой мыши на файле `App.config` или `Web.config` и выбрать пункт меню Edit WCF Configuration. Это хороший инструмент. Для этого примера приводится полученный XML-файл. Инструмент говорит сам за себя, и знание кода поможет как в обучении программированию, так и в выполнении сложного конфигурирования без редактора на более поздних этапах.

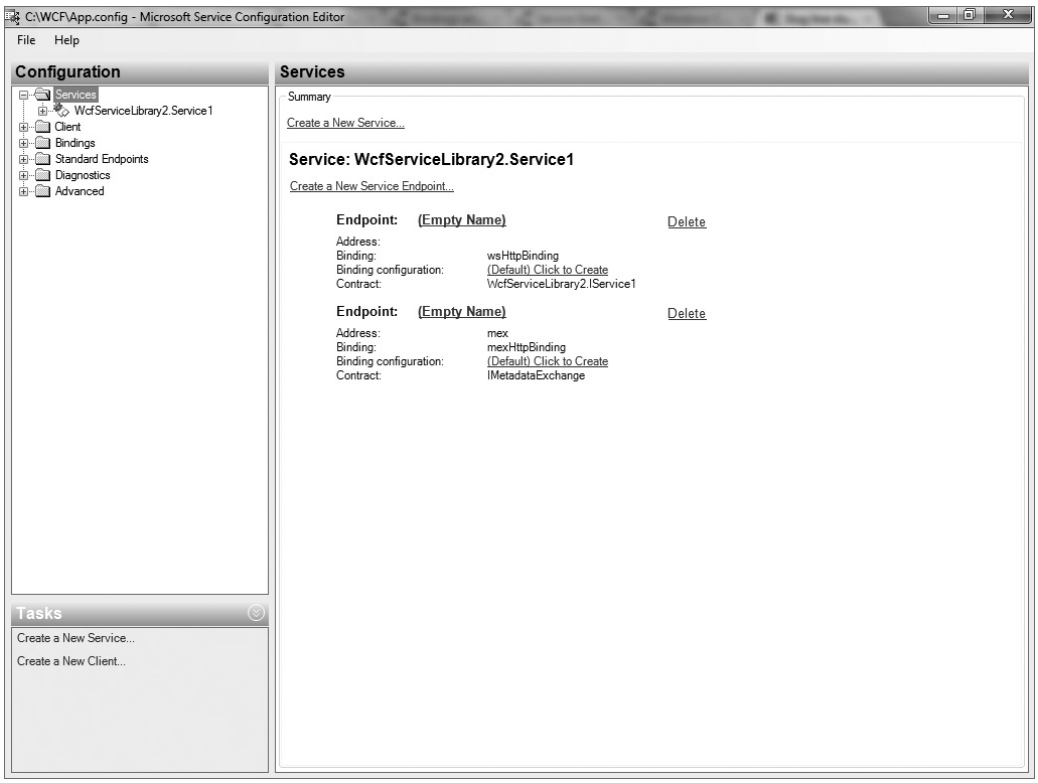


Рис. 3.2. Редактор конфигурации WCF Configuration Editor

В файле конфигурации есть специальный раздел, `system.serviceModel`, в котором вы можете определить все настройки реализации вашей службы или клиента: конечные точки, соединения, режимы, диагностики и др. Листинг 3.12 демонстрирует, как создать конечную точку в файле конфигурации, используя `wsHttpBinding` с контрактом службы `IMyService`.

Листинг 3.12. Декларативное конфигурирование соединения


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloWorldService">
        <endpoint
        address="http://localhost:8080/HelloWorldService"
        binding="wsHttpBinding"
        contract="IMyService" />
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
      </wsHttpBinding>
   </bindings>
  </system.serviceModel>
</configuration>
```

Как показано в листинге 3.13, вы можете получить такой же результат путем конфигурирования службы в программе.

Листинг 3.13. Программное конфигурирование службы


```
ServiceHost host = new ServiceHost(typeof(HelloWorldService));

wsHttpBinding binding = new wsHttpBinding();

host.AddServiceEndpoint(typeof(HelloWorldService),
                        binding,
                        "http://localhost:8731/HelloWorldService");
```

Базовый адрес

Нет необходимости точно указывать адрес в каждой конечной точке, можно указать базовый адрес в конфигурации, а затем использовать относительный адрес в каждой из конечных точек. Таким образом, можно получить множество конечных точек с одинаковым базовым адресом, т.е. если вы измените URL расположения службы, то сможете изменить его в одном месте, а не для каждой конечной точки. Возможно, это не покажется большим преимуществом, если у вас есть одна или две конечные точки, но если их много, то это быстро превращается в проблему. Это также спасает от многих орфографических ошибок в адресах и обеспечивает повторное использование.

В листинге 3.14 показано добавление относительного адреса в конечной точке к базовому адресу. По существу, это то же самое, что предыдущая конфигурация.

Листинг 3.14. Конфигурирование базового адреса в конфигурации


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloWorldService">
```

```
        <endpoint
        address="HelloWorldService"
        binding="wsHttpBinding"
        contract="IMyService" />
<host>
      <baseAddresses>
            <add baseAddress="http://localhost:8080/" />
      </baseAddresses>
</host>
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
      </wsHttpBinding>
   </bindings>
  </system.serviceModel>
</configuration>
```

Вы можете задать множество базовых адресов для каждой службы, но только один для каждого протокола. WCF автоматически отобразит правильный адрес в конфигурацию соединения в конечной точке.

В листинге 3.15 определены две конечные точки, одна из которых не имеет адреса, который использует `basicHttpBinding`. Это означает, что он будет отображен в путь базового адреса, определенного через HTTP. Вторая точка, которая имеет TCP, использующий `netTcpBinding`, имеет полный адрес `net.tcp://localhost:9090/tcp`.

Листинг 3.15. Несколько базовых адресов


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloWorldService">

        <endpoint
        address=""
        binding="basicHttpBinding"
        contract="IMyService" />
        <endpoint
        address="tcp"
        binding="netTcpBinding"
        contract="IMyService" />
<host>
      <baseAddresses>
            <add baseAddress="http://localhost:8080/" />
            <add baseAddress="net.tcp://localhost:9090/" />
      </baseAddresses>
</host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Конфигурация по умолчанию

Новая группа функций для WCF 4.0 предназначена для упрощения конфигурации и определяет конфигурацию по умолчанию для нескольких аспектов WCF. В общем, это позволяет уменьшить (или сделать ненужными) файлы конфигурации. Схемы конфигурации WCF — очень большие для настройки всех свойств. Ошибки также происходят, если службы распространяются с отсутствующей конфигурационной информацией. (Это особенно полезно для целей разработки.) Хотя это и не очевидно для очень простых служб, но когда у вас есть службы большого предприятия со многими конечными точками и многими различными настройками конфигурации для каждой, редактирование этих параметров становится тяжелой перспективой для разработчиков и администраторов. (Это особенно актуально для администраторов, которые не обязательно знают последствия изменения этих файлов конфигурации, но несут ответственность за них на производственных серверах.)

Следующие изменения являются новыми для WCF 4.0 и будут передавать исключения во время выполнения при использовании этих конфигураций в версиях ниже 4.0. Однако при обновлении приложений WCF 3/3.5 до WCF 4,0 они все равно работают с конфигурацией WCF 3.x. Но вы можете пересмотреть файл конфигурации и воспользоваться преимуществами ее упрощения, если хотите.

Автоматические конечные точки

Если вы не конфигурируете ни один из атрибутов `<service>` в вашем файле конфигурации (или файла конфигурации вообще нет) и ни одна конечная точка в программе не добавляется к хосту, то к вашей службе конечные точки будут добавлены автоматически. По одной точке будет сконфигурировано для каждой службы и каждого соглашения, а адрес конечной точки будет добавлен к базовому адресу.

При размещении вашей службы на IIS список базовых адресов автоматически извлекается из настроек хостинга. Если веб-сайт позволяет использовать как протокол HTTP, так и протокол HTTPS, вы обнаружите два адреса. Если на IIS применяется Windows Activation Services (WAS), то можно использовать протоколы, отличные от HTTP. Кроме того, если веб-сайт сконфигурирован для поддержки протокола NET.TCP, окажется, что доступ к вашей службе осуществляется через относительные конечные точки.

Различные возможности хостинга обсуждаются в главе 14.

Соглашения по умолчанию

Соглашения по умолчанию позволяют создать службу, которая автоматически устанавливает правильное соединение, вычисляя его из схемы протокола, которую вы указали в адресе службы (или в базовом адресе). Это делается с помощью отображения протокола.

WCF имеет предопределенное отображение протокола, который отображает схему протокола в соединение WCF. Значения по умолчанию приведены в табл. 3.3.

Таблица 3.3. Отображения протокола WCF по умолчанию

Схема	Соединение WCF
HTTP	`basicHttpBinding`
net.tcp	`netTcpBinding`
net.msmq	`netMsmqBinding`
net.pipe	`netNamedPipeBinding`

Вы можете изменить отображения, предлагаемые по умолчанию WCF 4.0, внеся добавления в файл конфигурации службы и изменив первоначальные установки. Тем самым изменения будут внесены исключительно в вашу программу. Если же вы хотите, чтобы изменения коснулись компьютера, сделайте то же в конфигурации компьютера (листинг 3.16).

Хитом по производительности будет ситуация, когда служба с настройками соединения в файле конфигурации запускает протокол, отображенный в вашем `machine.config`. Поступайте так с каждой конфигурацией или добавляйте все конфигурации, которые могут потребоваться для конфигурации вашей службы.

Листинг 3.16. Отображения протокола, предлагаемые WCF 4.0 по умолчанию

Доступно для загрузки на Wrox.com

```
<system.serviceModel>
    <protocolMapping>
        <add scheme="http" binding="basicHttpBinding"/>
        <add scheme="net.tcp" binding="netTcpBinding"/>
        <add scheme="net.pipe" binding="netNamedPipeBinding"/>
        <add scheme="net.msmq" binding="netMsmqBinding"/>
    </protocolMapping>
</system.serviceModel>
```

Вы можете изменить любой из этих параметров или добавить свои собственные. Как показано в листинге 3.17, если вы знаете, что все ваши службы будут использовать `WebHttpBinding`, можете изменить протокол отображения для HTTP (`basicHttpBinding`by по умолчанию) на `WSHttpBinding`.

Листинг 3.17. Изменение протокола отображения по умолчанию для транспортного протокола HTTP

Доступно для загрузки на Wrox.com

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="http" binding="wsHttpBinding"/>
  </protocolMapping>
</system.serviceModel>
```

В листинге 3.18 создается отображение между HTTPS и `WebHttpBinding`.

Листинг 3.18. Изменение протокола отображения для протокола HTTPs

Доступно для загрузки на Wrox.com

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="https" binding="wsHttpBinding"/>
  </protocolMapping>
</system.serviceModel>
```

Режимы по умолчанию

Режимы по умолчанию позволяют избежать явного указания режима, который вы хотите применить. Для этого можно воспользоваться атрибутом `behaviorConfiguration` в файле конфигурации.

В листинге 3.19 показаны изменения, внесенные в файлы конфигурации.

Листинг 3.19. Конфигурация без режима по умолчанию


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="MyService"
             behaviorConfiguration="MyServiceBehavior">
        <endpoint address="http://localhost/MyService/"
                binding="wsHttpBinding"
                contract="IMyService" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MyServiceBehavior">
        <serviceMetadata httpGetEnabled="True" />
     </behavior>
       </serviceBehaviors>
      </behaviors>
    </system.serviceModel>
  </configuration>
```

Обратите внимание на явное указание имени режима. В листинге 3.20, напротив, показано, как написать режим по умолчанию без указания имени.

Листинг 3.20. Конфигурация с режимом по умолчанию


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <system.serviceModel>
    <behaviors>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled ="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
 </system.serviceModel>
</configuration>
```

Стандартные конечные точки

Стандартные конечные точки допускают предопределенный набор конечных точек, и WCF предлагает несколько таких наборов. Некоторые из этих конечных точек чем-то похожи на все службы, — MEX, например, имеет такое же соглашение и соединение. Также можно создавать собственные конечные точки.

Ниже приведены стандартные конечные точки.

- mexEndpoint
- webHttpEndpoint
- webScriptEndpoint
- workflowControlEndpoint
- announcementEndpoint
- discoveryEndpoint
- udpAnnouncementEndpoint
- udpDiscoveryEndpoint

Чтобы использовать эти стандартные конечные точки, можно использовать атрибут kind в пределах обычной конечной точки, как показано в листинге 3.21.

Листинг 3.21. Стандартная конечная точка


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="MyService1">
        <endpoint kind="mexEndpoint">
      </service>
    </services>
  </system.serviceModel>
<configuration>
```

Настройка множественных конечных точек

Часто встречаются клиенты, подключающиеся к службам со многих различных типов систем и приложений, в частности WCF, ASMX и JSON. Вы можете указать несколько конечных точек с различными соединениями, чтобы различные клиенты могли подключаться к вашим службам с помощью любого поддерживаемого ими транспорта.

В листинге 3.22 добавляются две конечные точки с различными соединениями: wsHttpBinding и basicHttpBinding.

Листинг 3.22. Декларативное конфигурирование множества соединений


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloWorldService">
        <endpoint
        address="http://localhost:8080/HelloWorldService"
        binding="wsHttpBinding"
        contract="IMyService" />
        <endpoint
        address="http://localhost:8080/HelloWorldService/basic"
```

```
        binding="basicHttpBinding"
        contract="IMyService" />
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
      </wsHttpBinding>
   </bindings>
  </system.serviceModel>
</configuration>
```

Отметьте два отличия в объявлении этих конечных точек: первая, очевидно, соединение, и именно так и должно быть. Кроме того, адреса разные. Помните, что адреса должны быть уникальными. Вы должны создать новый адрес для `BasicHttpBinding`, чтобы использовать его. Если вы добавили `netTcpBinding`, новое соединение должно иметь уникальный адрес, поскольку схема протокола вначале больше не будет `http://`, а будет заменена на `net.tcp://`. Кроме того, если вы используете другое соглашение, конечная точка может иметь тот же самый адрес и то же самое соединение. Если адрес не уникален, а соглашение остается тем же, происходит исключение.

Модификация соединений

Несмотря на то что WCF предоставляет большое количество соединений, которые, как можно надеяться, соответствуют вашим потребностям в большинстве случаев, иногда необходимо создать собственный сценарий. Чтобы сделать это, можно "подогнать" один из предлагаемых сценариев и использовать свойства соединения или создать собственное пользовательское соединение. Если можно использовать существующее соединение, внеся небольшие изменения в свойствах, то это самый простой и предпочтительный способ. Вместе с тем WCF предлагает очень простой путь создания собственного соединения с использованием встроенных элементов соединения, которые являются строительными блоками каждого предлагаемого соединения.

Свойства соединений

Встроенные в соединения WCF имеют установленные по умолчанию свойства, которые можно модифицировать в соответствии с вашим сценарием. Все соединения имеют аналогичные свойства, но отличаются друг от друга.

В приведенном примере используется соединение `basicHttpBinding`, которое используется для коммуникации с веб-службой ASMX. В табл. 3.4 приведены свойства `basicHttpBinding` и кратко описано значение каждого свойства по умолчанию.

Таблица 3.4. Свойства `basicHttpBinding`

Свойство	Описание	Значение по умолчанию
`AllowCookies`	Указывает, допускает ли клиент файлы cookie	Ложь `False`
`BypassProxyOnLocal`	Должно быть установлено равным `true`, чтобы обходить прокси-сервер для локальных адресов	Ложь `False`
`CloseTimeout`	Промежуток времени, в течение которого соединение находится в состоянии ожидания перед закрытием по тайм-ауту	1 минута

Окончание табл. 3.4

Свойство	Описание	Значение по умолчанию
HostNameComparisonMode	Устанавливает, будет ли проверяться соответствие имени хоста службе	"StrongWildCard" (игнорирует имя хоста)
MaxBufferPoolSize	Устанавливает максимальный размер буфера для получения сообщений	65 536 байтов
MaxReceivedMessageSize	Устанавливает максимальный размер сообщения для данного соединения	65 536 байтов
MessageEncoding	Устанавливает используемую кодировку	Text
Name	Имя соединения	null
Namespace	Устанавливает пространство имен XML	http://tempura.org/
OpenTimeout	Устанавливает максимальное время открытия соединения до тайм-аута	1 минута
ProxyAddress	Адрес прокси-сервера	Null
ReceiveTimeout	Максимальное время до окончания по тайм-ауту операции получения сообщения	00:10:00
SendTimeout	Максимальное время до окончания по тайм-ауту операции отправки сообщения	00:10:00
TextEncoding	Кодировка текста сообщений	UTF8
TransferMode	Устанавливает режим передачи сообщений (буферный или потоковый)	Буферный
UseDefaultWebProxy	Устанавливает, должен ли использоваться системный прокси-сервер	Истина true

Изменение значения свойства MessageEncoding, равного по умолчанию Text, на Mtom. Mtom (*MessageTransmission Optimization Mechanism*) позволяет уменьшить размер сообщения при отправке больших объемов данных, отправляя сообщения SOAP в качестве сырых байт. Существенная разница в производительности в результате такого изменения весьма заметна.

Чтобы изменить это свойство, вначале создайте новую секцию конфигурации в секции basicHttpBinding. Она содержит имя для обращения. Затем установите значение свойства (одного или нескольких), которое нужно изменить. Потом к нему можно обращаться по новому имени в конфигурации конечной точки (в следующем примере это имя Encode). Это можно сделать, используя свойство bindingConfiguration в конфигурации конечной точки. Затем можно повторно использовать эту конфигурацию для других конечных точек в вашей конфигурации или, что может быть важнее, не использовать эту конфигурацию для других конечных точек. Это дает полную свободу в использовании конфигурации, но в то же время минимизирует дублирование свойств, допуская повторное использование. В листинге 3.23 показано, как можно изменить значения свойств по умолчанию в app.config.

Листинг 3.23. Изменение значений свойств по умолчанию в App.config

Доступно для загрузки на Wrox.com

```
<system.serviceModel>
    <services>
      <service name="MyService1">
        <endpoint address="http://localhost:8080/MyService1"
                contract="Service1"
```

```
            binding="basicHttpBinding"
            bindingConfiguration="Encode">
        </endpoint>
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name="Encode" messageEncoding="Mtom">    </binding>
      </basicHttpBinding>
    </bindings>
  </system.serviceModel>
```

То же самое можно сделать программно при создании нового объекта `basicHttpBinding`. Как показано в листинге 3.24, можно изменить свойства этого объекта так же, и как свойства любого другого объекта. Это также демонстрирует красоту средства IntelliSense из Visual Studio, так что вы можете увидеть все возможности конфигурируемого соединения.

Листинг 3.24. Изменение значений свойств по умолчанию программными средствами


```
ServiceHost host = new ServiceHost(typeof(MyService1.Service1));

BasicHttpBinding binding = new BasicHttpBinding();
binding.MessageEncoding = "Mtom";

host.AddServiceEndpoint(typeof(MyService1.IService1),
                        binding,
                        "http://localhost:8080/MyService1");
```

При редактировании `App.config` необходимо явно задать используемый `bindingConfiguration`, но в коде вы изменяете только текущий экземпляр созданного вами класса `BasicHttpBinding`.

Создание пользовательских соединений

Бывают ситуации, когда нужно создать свои собственные соединения; несколько примеров демонстрируют защиту данных пользователя и дополнительные транспортные протоколы.

Для того чтобы создать пользовательское соединение, создайте коллекцию элементов соединения. Вы должны быть хорошо знакомы с элементами соединения, определенными в WCF. Это позволит указать специфику вашего соединения.

Элементы соединения

Элементы соединения являются потомками `System.ServiceModel.Channels.BindingElement`. При добавлении элементов соединения вы должны соблюдать их последовательность. В табл. 3.5 приводятся предлагаемые соединения в той последовательности, в которой они должны быть добавлены в коллекцию соединений.

Таблица 3.5. Элементы протокола

Элементы соединения протокола	Имя класса
Выполнение транзакций	`TransactionFlowBindingElement`
Надежность обмена сообщениями	`ReliableSessionBindingElement`
Безопасность	`SecurityBindingElement`
Элемент двунаправленной коммуникации	
Смешанная двунаправленная	`CompositeDuplexBindingElement`
Элементы кодирования сообщений	
Кодирование сообщений	**Имя класса**
Текстовое	`TextMessageEncodingBindingElement`
Mtom	`MtomMessageEncodingBindingElement`
Двоичное	`BinaryMessageEncodingBindingElement`
Элементы безопасности транспорта	
Безопасность	**Имя класса**
Windows	`WindowsStreamSecurityBindingElement`
SSL	`SslStreamSecurityBindingElement`
Элементы транспорта	
Протокол	**Имя класса**
Http	`HttpTransportBindingElement`
Https	`HttpsTransportBindingElement`
TCP	`TcpTransportBindingElement`
Named pipes	`NamedPipeTransportBindingElement`
MSMQ	`MsmqTransportBindingElement`
MSMQ	`MsmqIntegrationBindingElement`
P2P	`PeerTransportBindingElement`

Для создания пользовательского соединения необходимо выбрать включаемые элементы соединения, а затем добавить их в пользовательское соединение в правильном порядке, как было сказано выше. В листинге 3.25 показано, что пользовательское соединение должно иметь транспортный элемент и элемент кодирования сообщений (даже если вы не указали кодировку сообщений, WCF автоматически добавит текст для HTTP и двоичный формат для всех остальных протоколов передачи).

Листинг 3.25. Пользовательское соединение с использованием HTTP для передачи и двоичного кодирования

Доступно для загрузки на Wrox.com

```
<customBinding>
  <binding name="binHttp">
      <binaryMessageEncoder />
      <httpTransport />
  </binding>
</customBinding>
```

Как пояснялось в начале этой главы, порядок элементов соединения гораздо важнее. Если считать снизу, первый элемент должен определять транспортный протокол. Второй элемент соединения должен определять используемую кодировку. Порядок остальных элементов не настолько важен, как двух первых.

Если не хотите использовать файл конфигурации, то можете создать пользовательское соединение программно, как показано в листинге 3.26.

Листинг 3.26. Конфигурирование пользовательского соединения в коде программы


```
using Wrox.CarRentalService.Implementations.Europe;
...

CustomBinding binding = new CustomBinding();
binding.Elements.Add(new BinaryMessageEncodingBindingElement());
binding.Elements.Add(new HttpTransportBindingElement());

ServiceHost host = new ServiceHost(typeof(CarRentalService));

ServiceEndpoint serviceEndpoint =
host.AddServiceEndpoint(typeof(ICarRentalService),
                        binding,
                        "http://localhost:8080/CarRentalService");

host.Open();
```

В данном коде можно просто добавить элемент соединения в коллекцию элементов пользовательского соединения. Таким образом определяется, как работает пользовательское соединение.

Повторно используемые пользовательские соединения

Иногда необходимо создать пользовательское соединение и использовать его повторно в различных приложениях. Поскольку каждое соединение является потомком базового абстрактного класса `System.ServiceModel.Channels.Binding`, можно просто создать собственный класс соединения и реализовать `CreateBindingElements`, чтобы получить то, что необходимо. В листинге 3.27 показано, как можно расширить базовый класс `Binding` для создания и повторного использования собственной реализации.

Листинг 3.27. Создание пользовательского повторно используемого соединения


```
public class NetTcpTextBinding : Binding
{
    private TcpTransportBindingElement transport;
    private TextMessageEncodingBindingElement encoding;

    public NetTcpTextBinding()
        : base()
    {
        this.Initialize();
    }

    public override BindingElementCollection CreateBindingElements()
    {
        BindingElementCollection elements = new BindingElementCollection();
```

```
        elements.Add(this.encoding);
        elements.Add(this.transport);
        return elements;
    }

    public override string Scheme
    {
        get { return this.transport.Scheme; }
    }

    private void Initialize()
    {
        this.transport = new TcpTransportBindingElement();
        this.encoding = new TextMessageEncodingBindingElement();
    }
}
```

В этом примере соединение `NetTcpTextBinding`, которое приведено в листинге 3.28, является пользовательским соединением, кодирующим сообщения с использованием протокола кодирования текста и транспорта TCP. Это очень простая реализация, демонстрирующая, как можно настроить WCF для получения того, что вам необходимо.

Как и любое другое соединение, можете использовать его с `ServiceHost` или `ChannelFactory`, чтобы конфигурировать конечную точку службы.

Листинг 3.28. Использование `NetTcpTextBinding`


```
NetTcpTextBinding binding = new NetTcpTextBinding();

ServiceHost host = new ServiceHost(typeof(Service1));
host.AddServiceEndpoint(typeof(IService1),
                        binding,
                        "net.tcp://localhost:10101/IService");

host.Open();
```

Следующий шаг позволяет разработчикам использовать пользовательское соединение через файл конфигурации. Как показано в листинге 3.29, вы должны расширить базовый абстрактный класс `BindingCollectionElement` и затем реализовать требуемые методы.

Листинг 3.29. Реализация `NetTcpTextBindingCollectionElement`


```
using System.Configuration;
...

public class NetTcpTextBindingCollectionElement : BindingCollectionElement
{
    public override Type BindingType
    {
        get { return typeof(NetTcpTextBinding); }
    }
```

```
    public override ReadOnlyCollection  <IBindingConfigurationElement>
    ConfiguredBindings
    {
        get
        {
            return new ReadOnlyCollection <IBindingConfigurationElement> (
                       new List <IBindingConfigurationElement> ());
        }
    }

    public override bool ContainsKey(string name)
    {
        throw new NotImplementedException();
    }

    protected override Binding GetDefault()
    {
        return new NetTcpTextBinding();
    }

    protected override bool TryAdd(
        string name, Binding binding, Configuration config)
    {
        throw new NotImplementedException();
    }
}
```

Теперь нужно реализовать свойство `BindingType`. Оно позволяет определить тип соединения в зависимости от текущей конфигурации. Другое важное свойство — `ConfiguredBindings`, которое возвращает все элементы конфигурации соединения.

Код, приведенный в листинге 3.29, является в самом деле простой реализацией, которая может быть расширена, например, определением элементов пользовательского соединения. Чтобы обеспечить высокий уровень настройки во время выполнения, необходимо реализовать интерфейс `IBindingConfigurationElement`.

Теперь, если хотите использовать `NetTcpTextBinding` в файле конфигурации, необходимо конфигурировать `NetTcpTextBindingCollectionElement`, используя расширения соединения (листинг 3.30). Конечно, если вы хотите использовать реализацию пользовательского соединения, вы должны определить ее.

Листинг 3.30. Использование `NetTcpTextBinding` в файле конфигурации


```
<system.serviceModel>
  <services>
    <service name="WcfServiceLibrary2.Service1">
      <endpoint address="net.tcp://localhost:10101/IService"
                binding="netTcpTextBinding"
                contract="WcfServiceLibrary2.IService1">

      </endpoint>
    </service>
  </services>
  <extensions>
```

```
    <bindingExtensions>
      <add name="netTcpTextBinding"
           type="ConsoleApplication1.NetTcpTextBindingCollectionElement,
           ConsoleApplication1,
           Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    </bindingExtensions>
  </extensions>
</system.serviceModel>
```

Значение атрибута `name`, определенного в секции расширения соединения, совпадает с именем, используемым в элементе `service`. Наконец, вы можете использовать соединение в элементе `endpoint` секции `service`.

Двунаправленные устойчивые службы

Ранее в этой главе обсуждались контекстные соединения. Представленные в .NET Framework 3.5, контекстные соединения позволяют передавать свойства, которые называются `instance id` и используются для идентификации сохраненных служб. Эта возможность позволяет службе помнить свой статус с помощью длительного и надежного хранилища, которое обрабатывает неожиданные сценарии, такие как перезапуск хоста или службы. Это важное отличие устойчивых служб от сессий, имеющихся в .NET Framework 3.0.

Уже довольно распространены сценарии, в которых процессы могут сохранять свое состояние в течение длительного времени. Windows обеспечивает поддержку длительных сценариев, сохраняя их состояние в базе данных, например в SQL Server. Устойчивые службы позволяют сохранять состояние службы, подобно хранению состояния исполняемой программы.

В .NET Framework 4.0, `WSHttpContextBinding` и `NetTcpContextBinding` имеют новое свойство `ClientCallbackAddress`. Это свойство получает URI, представляющее собой адрес клиента, используемый для получения обратного сообщения от вызванной службы. Как показано на рис. 3.3, если первый поток задач вызывает второй поток, а второй поток выполняется в течение длительного времени, первый поток запоминается. Теперь `ClientCallbackAddress` с контекстным соединением позволяет снова запустить первый процесс.

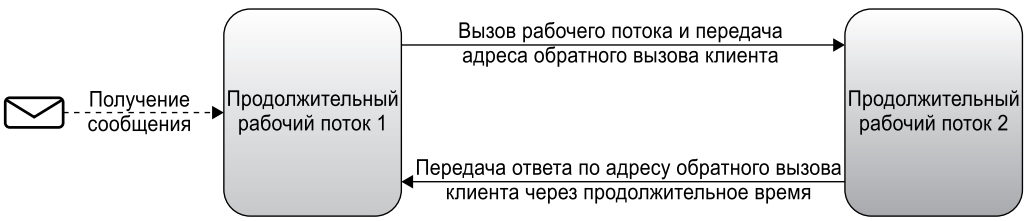


Рис. 3.3. Взаимодействие рабочих потоков

Сейчас двунаправленное устойчивое соединение работает иначе, чем уже известное двунаправленное соединение, потому что работает на уровне контекстных свойств и памяти, а не канала.

Конфигурирование соединений для служб с двунаправленным устойчивым соединением

Не все контекстные соединения поддерживают двунаправленное устойчивое согласование служб. В действительности `BasicHttpContextBinding` не поддерживает свойство `ClientCallbackAddress` — на самом деле поддерживается как `WSHttpContextBinding`, так и `NetTcpContextBinding`.

Чтобы разрешить согласование, нужно задать `ClientCallbackAddress` в конфигурации контекстного соединения, как показано в листинге 3.31.

Листинг 3.31. Конфигурирование соединения для получения устойчивого двунаправленного согласования


```
    <  system.serviceModel  >
      <  services  >
        <  service name="WorkflowService1"  >
          <  endpoint address=""
                  binding="wsHttpContextBinding"
                  contract="IWorkflowService"
                  bindingConfiguration="contextCorrelationBinding">
        </endpoint>
        <endpoint address=""
                  binding="wsHttpContextBinding"
                  contract="IWorkflowServiceCallback"
                  bindingConfiguration="contextCorrelationBinding">
        </endpoint>
      </service>
    </services>
    <bindings>
      <wsHttpContextBinding>
        <binding name="contextCorrelationBinding"
clientCallbackAddress="http://localhost/DurableDuplex/
WorkflowService1.xamlx">
        </binding>
      </wsHttpContextBinding>
    </bindings>
  </system.serviceModel
```

В предыдущем разделе конфигурации были определены две конечные точки: одна для службы и одна для обратного вызова. Оба элемента используют `WSHttpContextBinding` и значение свойства `clientCallbackAddress` в соответствующей секции под названием `contextCorrelationBinding`. Это единственный необходимый параметр для задания согласования.

Также можно создать пользовательское соединение и определить элемент контекстного соединения, как показано в листинге 3.32.

Листинг 3.32. Создание контекстного соединения с допустимым согласованием контекста


```
  < bindings >
      < customBinding >
          < binding name="netMsmqContextBinding" >
```

```
            < context clientCallbackAddress="net.msmq://localhost/private/
WorkflowService1.xamlx"/ >
            < msmqTransport / >
         < /binding >
      < /customBinding >
  < /bindings >
```

Контекстное соединение и система согласования является реализацией Microsoft Context Exchange Protocol (`http://msdn.microsoft.com/en-us/library/bb924468.aspx`).

`PollingDuplexHttpBinding`: опрос http

Изменения потребностей пользователей приводит к необходимости создания все более интерактивных веб-сайтов. В этом контексте Silverlight предлагает мощную платформу разработки для создания многофункциональных интерактивных приложений. Среда Silverlight работает как клиент в процессе браузера пользователя. Как и любые клиенты, он должен получить дистанционный доступ к данным, часто представленным как служба. Более удобной и очевидной платформой для разработки этих служб является WCF.

Наиболее распространенный сценарий допускает модель "вопрос–ответ" для обмена сообщениями, в которой клиент Silverlight посылает запрос и получает ответ от службы. Но иногда службе необходимо сообщить или предупредить клиента, что на другом конце канала передачи что-то происходит. Служба должна передавать данные клиенту. `PollingDuplexHttpBinding`, доступный начиная с Silverlight v3 в `System.ServiceModel.PollingDuplex.dll`, разработан для выполнения такого протокола коммуникации.

`PollingDuplexHttpBinding` является реализацией WS-MakeConnection v1.1 OASIS Specification (`http://docs.oasis-open.org/ws-rx/wsmc/v1.1/wsmc.html`). На практике, когда сессия установлена, клиент посылает запрос к службе, и служба реагирует HTTP-ответом только в том случае, когда ей необходимо отправить данные клиенту, т.е. когда имеются данные, предназначенные для клиента. Это делается с помощью механизма, уже известного как *долговременный опрос*. Если для клиента нет данных, служба сохраняет последний HTTP-запрос до тех пор, пока не появятся данные или до истечения времени ожидания. Если клиенту необходима дополнительная информация, он отправляет новый HTTP-запрос. Коммуникация продолжается до тех пор, пока поддерживается HTTP-запрос.

На стороне клиента вы должны обращаться к клиентской версии `PollingDuplexHttpBinding` из `System.ServiceModel.PollingDuplex.dll`, которая находится в той же папке, что и библиотека Silverlight v3 SDK. Затем сконфигурируйте соединение, как показано в листинге 3.33.

Листинг 3.33. Конфигурирование `PollingDuplexBinding` в среде клиента


```
PollingDuplexHttpBinding binding = new PollingDuplexHttpBinding()
{
    InactivityTimeout = TimeSpan.FromMinutes(10),
    ReceiveTimeout = TimeSpan.FromMinutes(30),
    SendTimeout = TimeSpan.FromSeconds(30)
```

```
};
ChannelFactory  <  IService1  >   factory =
    new ChannelFactory<IService1>(binding, "http://localhost:10101/IService");
```

Также на стороне клиента можно просто использовать `PollingDuplexHttpBinding` для включения коммуникации, как показано в листинге 3.34.

Листинг 3.34. Конфигурирование `PollingDuplexBinding` на стороне службы


Доступно для загрузки на Wrox.com

```
PollingDuplexHttpBinding binding = new PollingDuplexHttpBinding();
ServiceHost host = new ServiceHost(typeof(Service1));
host.AddServiceEndpoint(typeof(IService),
                        binding,
                        "http://localhost:10101/IService");
host.Open();
```

`PollingDuplexHttpBinding` позволяет решать проблемы коммуникации и не требует дальнейших настроек коммуникации для хорошей работы. Как вы можете видеть, длительный опрос в HTTP отличается от традиционных систем длительного опроса, где обычно клиент посылает сообщения для выяснения состояния службы. Это приводит к увеличению трафика и иногда приводит к затору в сети. Для уменьшения этой проблемы множество обозревателей, например Internet Explorer 8, разрешает не более 6-ти соединений для каждого хоста.

Вы должны уделить внимание этому ограничению, несмотря на то, что технология долговременного опроса может дать большие преимущества. Она может легко превысить допустимый предел соединений обозревателя, если клиент Silverlight используется слишком интенсивно.

`NetTcpBinding` в Silverlight 4

В Silverlight 4 вы можете управлять двунаправленным соединением между клиентом Silverlight и службой WCF, используя также транспортный протокол net.tcp.

По сравнению с `PollingDuplexHttpBinding`, net.tcp в Silverlight 4 серьезно ухудшил производительность и может быть предпочтительным только в случае внутрикорпоративной сети (intranet). На самом деле протокол net.tcp налагает множество ограничений. Разрешается использовать только порты с 4502 по 4534, причем во внутрикорпоративной сети вы можете контролировать соответствие требованиям настроек системы защиты. Наконец, настройки безопасности передачи сообщений (transport security settings — SSL) в транспортном протоколе net.tcp в Silverlight не поддерживаются. Если нет необходимости в безопасности транспорта, данное ограничение не так важно.

Протокол net.tcp в Silverlight является большим шагом вперед и хорошей альтернативой HTTP-протоколу, хотя он имеет ограниченное применение из-за вышеупомянутых ограничений.

На момент написания этой книги Silverlight 4 только готовился к выпуску, и то, что обсуждается в этом разделе, может измениться в финальной версии.

4

Клиенты

В ЭТОЙ ГЛАВЕ...

- Реализация клиентов WCF
- Реализация и использование служб RESTful
- Предоставление вашей службы AJAX
- Использование WCF из Silverlight

Некоторые главы этой книги сосредоточены в основном на реализации стороны сервера, включая такие задачи, как разработка служб WCF, проектирование службы и контракта данных, выбор подходящего значения `InstanceContextMode`, `ConcurrencyMode` и, наконец, но не в последнюю очередь, использование подходящей привязки.

Основное внимание в этой главе уделяется стороне клиента. Вам уже, вероятно, встречались случаи, когда инфраструктура .NET использовалась на клиентской стороне, а для связи между приложением-клиентом и службой использовался прокси-сервер. Прокси-сервер может быть создан в окне Add Service Reference (Добавление ссылки на службу) или с помощью утилиты `svcutil.exe` (для приложений Silverlight — утилитой `SLsvcutil.exe`) и ссылки на файл WSDL или конечную точку MEX. Но если у вас есть неограниченный контроль над кодом сервера и клиента, вы обойдетесь без прокси-сервера и сможете непосредственно общаться со службой, используя ссылку на сборку службы и класс `ChannelFactory`.

В ориентированном на службы мире, где используются компоненты на очень разных платформах и с применением самых разнообразных технологий, ваши службы могут быть использованы другими технологиями, такими как ASP.NET AJAX, Java, PHP и Silverlight.

В зависимости от ожидаемых клиентов при определенных обстоятельствах может иметь смысл использовать не весь набор средств, предоставляемых платформой WCF.

Например, клиент AJAX может обработать сообщения в формате JSON намного быстрее, чем расточительные сообщения SOAP. Другие клиенты, в свою очередь, могут поддерживать очень немного протоколов WS*, если они вообще будут их поддерживать. В этих случаях имеет смысл договариваться о самом низком общем знаменателе (спецификация Basic Profile версии 1.1). И наконец, но не в последнюю очередь, стиль REST представляет собой архитектурную альтернативу тяжеловесному миру SOA с его протоколом SOAP, а также расточительными протоколами WS*, некоторые из которых весьма трудны в реализации. Стиль REST концентрируется на адресуемых ресурсах и их разнообразных характеристиках; для чтения и манипулирования данными он использует интуитивно понятный и единообразный интерфейс API.

В следующих разделах каждый из этих аспектов рассматривается более подробно.

Спецификация Basic Profile 1.1

Для гарантии корректной работы связи с использованием спецификации Basic Profile версии 1.1 между различными системами участвующие стороны должны соблюдать определенные правила.

Над стандартизацией протоколов, форматов и определений работает множество учреждений. В приложениях веб-служб спецификация Basic Profile 1.1 — самый нижний общий знаменатель, когда дело доходит до обмена сообщениями, предоставления метаданных, описания служб и определений безопасности. Спецификация Basic Profile 1.1 опубликована организацией по совместимости веб-служб, Interoperability Organization (WS-I), по адресу `http://www.ws-i.org/Profiles/BasicProfile-1-1.html`. К описанным и рекомендуемым стандартам относятся SOAP 1.1, WSDL 1.1, UDDI 2.0, XML 1.0 и некоторые другие.

Чтобы сделать службу WCF совместимой со спецификацией Basic Profile 1.1, достаточно использовать привязку `basicHttpBinding`.

В платформу WCF 4.0 добавлена также поддержка спецификации Basic Profile 1.2.

Но если ваша служба должна использовать такие передовые технологии, как транзакции и специальные механизмы защиты, то необходимо применять соответствующий протокол WS*. Вы, конечно, подвергнетесь риску, но ваша служба сможет использоваться весьма специфическими клиентами.

Клиенты .NET

Когда дело доходит до реализации клиентов служб среды WCF на платформе .NET, можно использовать различные методики. Можно использовать предоставляемый документ WSDL, чтобы создать прокси-сервер и использовать его для связи со службой. Либо ваш клиент .NET может совместно использовать со службой ту же сборку контракта данных. Позже мы рассмотрим некоторые из этих возможностей более подробно.

Совместное использование контрактов WSDL

Если вы собираетесь использовать службу, предоставляемую неким провайдером служб, в ориентированном на службы мире, то вам обычно не предоставляется никаких библиотек DLL и никаких других файлов бинарного кода. Вместо этого все клиенты и

сервер совместно используют документ WDSL, предоставляемый провайдером служб. Этот документ содержит все подробности, необходимые для взаимодействия со службой, включая структуру данных для обмена информаций, любые политики, такие как политика безопасности и надежности, а также информацию о том, как служба взаимодействует с дополнительными функциональными возможностями.

Этот вариант имеет смысл и рекомендуется для использования в не жестко связанной системе. Клиенту не требуется никакой информации о внутренней реализации, так как вся передача осуществляется исключительно на основе поставляемого документа WDSL. Поскольку клиент не имеет со службой никакого общего кода, это не обязательно должен быть клиент .NET. Это вполне может быть, например, приложение PHP или Java.

Если в качестве клиента используется приложение .NET, вы вообще не осуществляете непосредственного программирования, в отличие от предоставляемой службы; вы полагаетесь на функциональные возможности прокси-сервера. Прокси-сервер берет на себя такие задачи, как передача службе сообщений SOAP, получение ответа и предоставление его как объекта. В дополнение к предоставляемым службой операциям прокси-сервер WCF предоставляет также дополнительные функции, такие как открытие и закрытие, а также такие средства, как конечная точка и внутренний канал, чтобы читать идентификатор сеанса или специфическую информацию конечной точки.

Прокси-сервер обычно создается с помощью диалогового окна Add Service Reference (Добавление ссылки на службу), как показано на рис. 4.1, или инструмента `svcutil.exe`. Если вы находитесь в среде разработки Visual Studio, то проще и удобней воспользоваться меню Add Service Reference. Но если вы создаете прокси-сервер, то потребуется больше параметров, чем предоставляет диалоговое окно Add Service Reference (рис. 4.2). Когда необходимо экспортировать метаданные службы из сборки, можете также использовать инструмент `svcutil.exe` непосредственно.

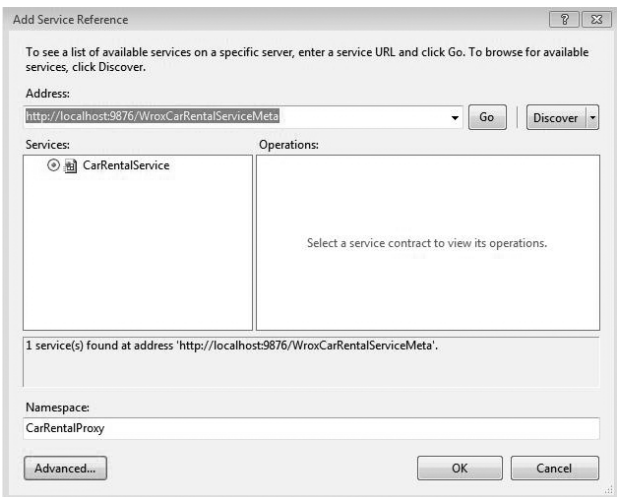


Рис. 4.1. Диалоговое окно Add Service Reference

Если вы добавите ссылку на службу, то получите два файла.

- Файл кода, используемый для связи со службой, включая прокси-сервер службы, классы данных и взаимодействия.
- Файл конфигурации, в котором определяется клиентская конечная точка.

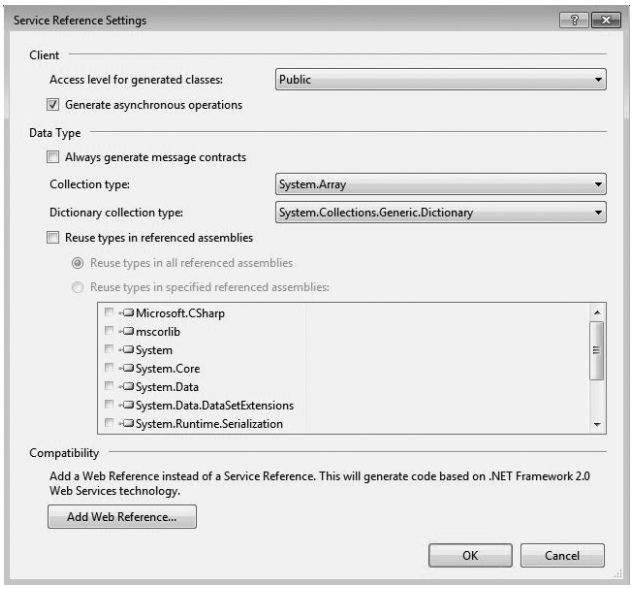


Рис. 4.2. Параметры диалогового окна Add Service Reference

Код, создаваемый утилитой `svcutil.exe`, расширяет базовый класс `System.ServiceModel.ClientBase<T>` и реализует интерфейс службы, а следовательно, и предоставляемые операции службы. Классы данных службы копируются также на сторону клиента и получают атрибуты `[DataContract]` и `[DataMember]`. В принципе класс `ClientBase<T>` является ничем иным, как оболочкой для класса `ChannelFactory<T>`, и представляет собой самый быстрый и простой способ общения со службой.

Например, код службы листинга 4.1 привел бы к предоставлению клиентам кода, приведенного в листингах 4.2–4.4.

Листинг 4.1. Контракт службы


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(
     Namespace = "http://wrox/CarRentalService/2009/10",
     Name = "RentalService")]
    public interface ICarRentalService
    {
        [OperationContract()]
        PriceCalculationResponse CalculatePrice
            (DateTime pickupDate, DateTime returnDate,
             string pickupLocation, string returnLocation);

        [OperationContract]
        Guid Confirm(Guid requestID);
```

```
        [OperationContract()]
        void ReportCrash
            (DateTime date, String location, Guid confirmationID);
    }
}
```

Листинг 4.2. Сторона клиента: интерфейс контракта службы


```
[System.ServiceModel.ServiceContractAttribute(
Namespace="http://wrox/CarRentalService/2009/10",
ConfigurationName="CarRentalProxy.RentalService")]
public interface RentalService {

[System.ServiceModel.OperationContractAttribute(
Action="http://wrox/CarRentalService/2009/10/RentalService/CalculatePrice",
ReplyAction="http://wrox/CarRentalService/2009/10/RentalService/
CalculatePriceResponse")]
Wrox.CarRentalService.ConsoleClient.CarRentalProxy.PriceCalculationResponse
    CalculatePrice(
        System.DateTime pickupDate, System.DateTime returnDate,
        string pickupLocation,  string returnLocation);

[System.ServiceModel.OperationContractAttribute(AsyncPattern=true,
Action="http://wrox/CarRentalService/2009/10/RentalService/CalculatePrice",
ReplyAction="http://wrox/CarRentalService/2009/10/RentalService/
CalculatePriceResponse")]
System.IAsyncResult BeginCalculatePrice
        (System.DateTime pickupDate, System.DateTime returnDate,
        string pickupLocation,  string returnLocation,
        System.AsyncCallback callback, object asyncState);

Wrox.CarRentalService.ConsoleClient.CarRentalProxy.PriceCalculationResponse
        EndCalculatePrice(System.IAsyncResult result);
```

Листинг 4.3. Сторона клиента: класс `ClientBase<T>`


```
public partial class RentalServiceClient :
        System.ServiceModel.ClientBase
            <Wrox.CarRentalService.ConsoleClient.CarRentalProxy.RentalService>,
            Wrox.CarRentalService.ConsoleClient.CarRentalProxy.RentalService
{
```

Листинг 4.4. Сторона клиента: файл конфигурации


```
<configuration>
    <system.serviceModel>
        <client>
            <endpoint address="http://localhost:9876/WroxCarRentalService"
                binding="basicHttpBinding"
                contract="CarRentalProxy.RentalService" />
        </client>
    </system.serviceModel>
</configuration>
```

Когда вы используете прокси-сервер так, как показано в листинге 4.5, не забывайте закрывать его, если он больше не нужен. Либо можете использовать прокси-сервер в операторе `using`, чтобы гарантировать в конце автоматический вызов метода `Dispose`, освобождающего ресурсы.

Листинг 4.5. Использование клиентского прокси-сервера


```
namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (CarRentalProxy.RentalServiceClient
                   rentalServiceClient = new
                   CarRentalProxy.RentalServiceClient())
            {
                CarRentalProxy.PriceCalculationResponse resp = null;
                resp = rentalServiceClient.CalculatePrice
                    (DateTime.Now, DateTime.Now.AddDays(5),
                     "Graz", "Vinna"),
                Console.WriteLine("Price to Vienna {0}",
                                  resp.Price);
            }
            Console.WriteLine("Proxy closed"),
        }
    }
}
```

Совместное использование контрактов WSDL и библиотек DLL `DataContract`

Если вы несете ответственность за реализацию как сервера, так и клиента, то имеете полный контроль над создаваемым кодом. Поэтому при определенных обстоятельствах может иметь смысл воспрепятствовать созданию новых классов данных при каждом добавлении ссылки на службу, что привело бы к дублированию и требовало бы дополнительных затрат. Чтобы воспрепятствовать этому, можно также совместно использовать контракт данных, чтобы ваш код на стороне сервера и на стороне клиента выглядел по-разному. Например, список `List<string>` на стороне сервера стал бы строковым массивом на стороне клиента. Некоторые аспекты позволяет также контролировать диалоговое окно Add Service Reference - Advanced (Добавление ссылки на службу — Дополнительно).

Чтобы указать утилите `svcutil.exe` использовать классы данных из библиотеки DLL, на которую имеется ссылка, и не создавать новые классы данных, передайте ей параметр `/r`. Еще одно преимущество использования общих классов данных заключается в том, что вы можете использовать классы данных любой сложности, обеспечивая им возможность сериализации.

Клиентскому коду все еще поставляется класс службы в форме прокси-сервера, который унаследован от класса `System.ServiceModel.ClientBase<T>` и реализует интерфейс службы. Однако классы данных службы больше не моделируются на стороне клиента и загружаются непосредственно из совместно используемой сборки.

Совместное использование интерфейсов и библиотек DLL `DataContract`

Самая простая форма привязки между клиентом и сервером не подразумевает использования документа WSDL вообще. Она подразумевает создание прямой ссылки на сборки `ServiceContract` или `DataContract`. Конечно, это возможно только в такой системе, где у вас есть неограниченный контроль над кодом сервера и клиента. Класс `ChannelFactory` используется для связи со службой, а не с прокси-сервером, как представлено в листинге 4.6. Объекты `ChannelFactory` используются, прежде всего, в сценариях промежуточного уровня, задача которых заключается в увеличении производительности, и где нет никакой необходимости в создании каждый раз нового экземпляра прокси-сервера для каждого клиента. Объект `ChannelFactory` используется для того, чтобы открыть отдельный канал для каждого клиента. Кроме того, в системах, в которых часто вносят изменения в контракт службы (в частности, в начале проекта), имеет смысл (и очень удобно) создавать код нового прокси-сервера каждый раз, а не использовать непосредственно сборку сервера.

Листинг 4.6. Класс `ChannelFactory<T>`


```
using System,
using System.Collections.Generic,
using System.Linq,
using System.Text,
using System.ServiceModel,
using Wrox.CarRentalService.Contracts,

namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            ChannelFactory<ICarRentalService> factory = null,
            try
            {
                BasicHttpBinding binding = new BasicHttpBinding(),
                EndpointAddress address = new
                EndpointAddress("http://localhost:9876
                                /WroxCarRentalService"),

                factory = new
                ChannelFactory<ICarRentalService>(binding, address),

                ICarRentalService channel = factory.CreateChannel(),

                PriceCalculationResponse resp =
                    channel.CalculatePrice
                        (DateTime.Now, DateTime.Now.AddDays(5),
                         "Graz", "Wien"),
                Console.WriteLine("Price to Wien {0}", resp.Price),

                factory.Close(),
            }
```

```
            catch (CommunicationException)
            {
                if (factory != null)
                {
                    factory.Abort(),
                }
            }
            catch (TimeoutException)
            {
                if (factory != null)
                {
                    factory.Abort(),
                }
            }
            catch (Exception ex)
            {
                if (factory != null)
                {
                    factory.Abort();
                }
                Console.WriteLine(ex.ToString());
            }

            Console.WriteLine("Proxy closed");
        }
    }
}
```

В примере на рис. 4.3 клиент непосредственно обращается к сборке службы и к сборке `DataContract`.

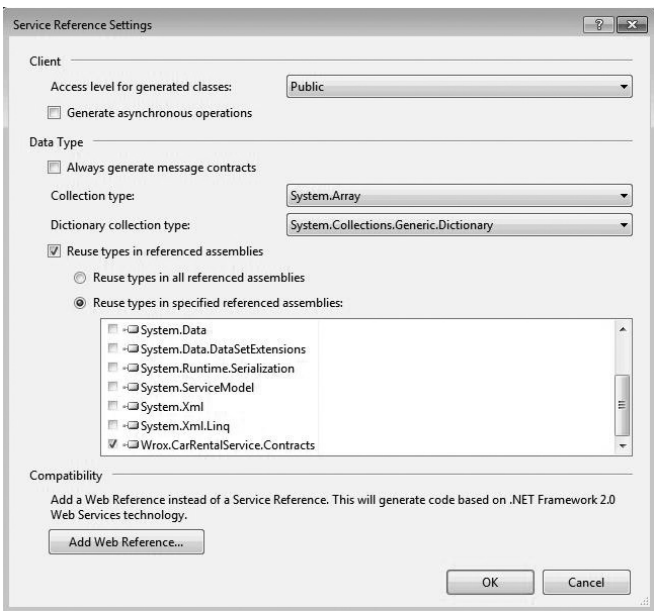


Рис. 4.3. Клиент непосредственно обращается к сборке службы и к сборке **`DataContract`**

Однако в ссылке на конкретную реализацию класса службы нет никакой необходимости, поскольку связь между клиентом и сервером все еще осуществляется через стандартный маршрут WCF.

Стиль REST

Термин REST (Representational State Transfer — репрезентативная передача состояния) впервые появился в диссертации Роя Томаса Филдинга (Roy Thomas Fielding).

Одним из важнейших аспектов стиля REST являются уникально адресуемые ресурсы, их различные характеристики и форматы, единообразный, доходчивый интерфейс программирования и помощь чрезвычайно высокомасштабируемой среды.

Нынешняя реализация стиля REST подразумевает, главным образом, использование хорошо устоявшихся в Интернете технологий, к которым все уже привыкли. Например, в качестве протокола передачи может использоваться протокол HTTP или HTTPS. Для обращения к ресурсам могут использоваться URL, включающие строки запроса, а форматы представления распространяются от протоколов HTML и XML до JSON и ATOM, а также файлы звука и видео. Упомянутый ранее простой и интуитивно понятный интерфейс программирования обеспечивается за счет использования команд протокола HTTP и кодов состояния.

Например, если вы переходите в браузере по URL `http://p2p.wrox.com/content/blogs`, возвращается код состояния 200 и ваш браузер отображает возвращенный код HTML в форме веб-сайта. Согласно обычным практикам веб, следующая страница также содержит ссылки на другие ресурсы и темы.

```
http://p2p.wrox.com/content/blogs/danm/announcing-virtualdnug
```

В URL, приведенном выше, заметно, насколько интуитивно понятна структура этих URL. Или используйте следующий URL, чтобы получить ленту RSS.

```
http://www.wrox.com/WileyCDA/feed/RSS_WROX_ALLNEW.xml
```

В отличие от архитектуры SOA с протоколом SOAP архитектура REST не сводится к определению сообщений и проектированию методов; здесь основные компоненты — это ресурсы и действия, способные влиять на эти ресурсы. Этими действиями, главным образом, являются методы CRUD (Create, Read, Update and Delete — создание, чтение, изменение и удаление). То, что должно быть сделано с ресурсом, определяется командой протокола HTTP, а успех действия устанавливается при запросе кода состояния протокола HTTP.

В следующем примере команда протокола `GET` по адресу `http://localhost:1234/CarPool` приведет к отображению информации обо всех автомобилях в форме документа XML. Код состояния 200 означает успех.

```
URL: http://localhost:1234/CarPool
Verb: GET
Status-Code: 200
Response-Body:
<Cars xmlns="http://schemas.datacontract.org/2004/07/Wrox"
xmlsn:i="http://www.w3.org/2001/XMLSchema-
instance"><Car><Make>Dodge</Make><Name>Dakota</Name><Seats>8</Seats><Type>Pickup

Truck</Type></Car><Car><Make>Audi</Make><Name>TT</Name><Seats>2</Seats><Type>Sport
Car</Type></Car><Car><Make>Seat</Make><Name>Leon</Name><Seats>5</Seats><Type>Sport
Car</Type></Car></Cars>
```

В следующем примере команда GET по адресу http://localhost:1234/CarPool/TT приведет к отображению информации об автомобиле "Ауди TT" в форме документа XML. Код состояния 200 означает успех, а код состояния 404, например, означает невозможность найти информацию.

```
http://localhost:1234/CarPool/TT
Verb: GET
Status-Code: 200
Response-Body:
<Car xmlns="http://schemas.datacontract.org/2004/07/Wrox"
xmlsn:i="http://www.w3.org/2001/XMLSchema-
instance"><Make>Audi</Make><Name>TT</Name><Seats>2</Seats><Type>Sport
Car</Type></Car>
```

В следующем примере команда GET по адресу http://localhost:1234/CarPool/TT?format=json приведет к отображению информации об автомобиле "Ауди TT" в форме документа JSON. Код состояния 200 означает успех, а код состояния 404 означает невозможность найти информацию.

```
URL: http://localhost:1234/CarPool/TT?format=json
Verb: GET
Status-Code: 200 (OK)
Response-Body:
{"Make":"Audi","Name":"TT","Seats":2,"Type":"Sport Car"}
```

В следующем примере команда PUT по адресу http://localhost:1234/CarPool/Leon с необходимыми данными XML в теле запроса означала бы, что информация об автомобиле должна быть добавлена к списку автомобилей. Код состояния 201 означает успех создания записи. В результате действия POST добавленный объект обычно возвращается назад вместе с соответствующими подробностями или ссылками на другие объекты.

```
URL: http://localhost:1234/CarPool/Leon
Verb: PUT
Request-Body:
<Car xmlns="http://schemas.datacontract.org/2004/07/Wrox"
 xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
<Make>Seat</Make><Name>Leon</Name><Seats>4</Seats><Type>Sport
Car</Type></Car>
Status-Code: 201 (Created)
Response-Body:
<Car xmlns="http://schemas.datacontract.org/2004/07/Wrox"
xmlsn:i="http://www.w3.org/2001/XMLSchema-
instance"><Make>Seat</Make><Name>Leon</Name><Seats>4</Seats><Type>Sport
Car</Type></Car>
```

Кроме уже продемонстрированных команд GET и PUT, чаще остальных используются команды DELETE и POST.

Команда GET используется исключительно для получения данных, поэтому результат может быть буферизирован без всяких проблем. Это также главное преимущество перед сообщениями SOAP, в которых фактический вызов не может быть немедленно распознан как запрос на чтение. Поэтому их буферизация затруднена.

Команда DELETE используется для удаления ресурса. По необходимости ее можно задействовать без проблем, не заботясь о побочных эффектах. Например, при первом

применении к адресу `http://localhost:1234/CarPool/Leon` информация об автомобиле "Леон" удаляется. При следующем запросе того же URL ресурс будет недоступен, а потому и не может быть удален.

Команда `PUT` используется для добавления или изменения ресурса, если вы можете определить URL самостоятельно. Например, команда `PUT` по адресу `http://localhost:1234/CarPool/Leon` наряду с соответствующими подробностями в теле запроса означает, что этот ресурс добавляется по используемому URL. Если использовать команду `PUT` для того же URL, но с другими значениями в теле запроса, то это приведет к изменению информации ресурса.

Команда `POST` — в некотором смысле исключение. Во-первых, ее нередко неправильно применяют вместо команд `DELETE` и `PUT`, поскольку их использование либо не разрешается, либо технически невозможно с точки зрения браузера. Во-вторых, команда `POST` используется для добавления новых ресурсов в контейнер без контроля над созданным адресом URL. Таким образом, например, команда `PUT` по адресу `http://localhost:1234/CarPool` с соответствующими подробностями в теле означала бы добавление нового автомобиля к базе. Но в данном случае вызов команды `POST` должен был бы возвратить URL недавно созданного ресурса.

В любом случае архитектура REST не поддерживает фиксации состояния. Поэтому всякий раз, когда осуществляется запрос, все необходимые значения передаются либо в адресе ресурса, либо как дополнительные параметры. Это поведение также означает, что приложения REST, как правило, существенно проще масштабировать, поскольку ненужно беспокоиться о том, на какой сервер послать запрос в среде с балансом нагрузки.

Архитектура REST и платформа WCF

Среда WCF не только предоставляет платформу для сообщений SOAP, она также является средством предоставления служб RESTful.

Основные компоненты, необходимые для создания приложения REST, находятся в библиотеке `System.ServiceModel.Web.dll`. К важнейшим элементам приложения REST относятся атрибуты `[WebGet]` и `[WebInvoke]`, а также новая привязка `webHttpBinding` совместно с режимом конечной точки `webHttp`.

Если вы используете атрибут `[WebGet]` или `[WebInvoke]`, атрибут `[OperationContract]` необязателен.

Если вы хотите создать решение RESTful из вашего решения на базе протокола SOAP, то теоретически должны были бы просто заменить привязку на `webHttpBinding` и настроить конечную точку на режим `webHttp`.

Эта привязка дает следующий эффект: если вы пошлете сообщение SOAP на конечную точку в классической среде SOA, то содержимое сообщения сначала будет проверяться. Корректная команда будет выполнена с учетом содержимого сообщения.

Но в архитектуре REST вызываемый метод зависит от URL и используемой команды. Такое изменение поведения диспетчеризации возможно благодаря привязке `webHttpBinding` и режиму `webHttp` конечной точки. Кроме того, класс `WebHttpBehavior` или режим конечной точки позволит также выбрать один из двух форматов сериализации.

- Формат POX (Plain old XML – старый простой XML) использует только формат XML без дополнительных затрат протокола SOAP.
- Формат JSON (JavaScript Object Notation – объектная нотация JavaScript) является очень компактным и эффективным форматом, особенно для соединений JavaScript.

Например, контракт службы, представленный в листинге 4.7, совместно с измененной конфигурацией, отображенной в листинге 4.8, привел бы к запросу `POST` по адресу `http://localhost:1234/HelloWorld` и получению документа XML "Hello World".

Листинг 4.7. Контракт службы


```
[ServiceContract()]
    public interface ICarRentalService
    {
        [OperationContract]
        string HelloWorld(),
```

Листинг 4.8. Файл конфигурации с привязкой `webHttpBinding` и режимом `webHttp`


```
<configuration>
    <system.serviceModel>
        <services>
        <service name="Wrox.CarRentalService">
            <endpoint
                address="http://localhost:1234/"
                contract="Wrox.ICarRentalService"
                binding="webHttpBinding"
                behaviorConfiguration="web"/>
            </service>
        </services>
        <behaviors>
            <endpointBehaviors>
                <behavior name="web">
                    <webHttp/>
                </behavior>
            </endpointBehaviors>
        </behaviors>
    </system.serviceModel>
</configuration>
```

Вместо элементов записей в файле конфигурации можете также использовать новый класс `WebServiceHost`. Класс `WebServiceHost` происходит от класса `ServiceHost` и автоматически назначает правильную привязку и режим вашей конечной точке. Следовательно, можете больше не беспокоиться о содержимом своего файла конфигурации и оставить свою службу вообще без файла конфигурации. В качестве альтернативы, если вы держите свою службу на сервере IIS, используйте класс `WebServiceHostFactory`. Например.

```
<% @ServiceHost Factory= "System.ServiceModel.Web.WebServiceHostFactory"
```

При серверном коде, представленном в листинге 4.9, файл конфигурации может оставаться пустым.

Листинг 4.9. Класс `WebServiceHost`


```
namespace Wrox.ConsoleHost
{
    class Program
    {
        static void Main(string[] args)
        {
            WebServiceHost webHost = null;
            try
            {
                webHost = new WebServiceHost(
                    typeof(Wrox.CarRentalService),
                    new Uri("http://localhost:1234")
                    );
                webHost.Open();
                Console.ReadLine();
            }
            catch (Exception ex)
            {
                if (webHost != null)
                    webHost.Abort();
                Console.WriteLine(ex.ToString());
                }
        }
    }
}
```

Но недостаток применения стандартной привязки `webHttpBinding` совместно с режимом `enableWeb` заключается в том, что, во-первых, команда `POST` используется как стандартная команда и, во-вторых, используемый URL является просто названием метода. Но это не отражает основную идею архитектуры REST. В отличие от архитектуры SOA здесь главное значение имеют не сообщения, а ресурсы, URL и используемые команды.

Получить полный контроль над используемым URL можно с помощью двух атрибутов, `[WebGet]` и `[WebInvoke]`, в комбинации с атрибутом `OperationContract` (ныне устаревшим).

Используйте атрибут `[WebGet]` всякий раз, когда необходим доступ к ресурсу для чтения, т.е. когда применяется команда протокола HTTP `GET`.

Используйте атрибут `[WebInvoke]` для всех остальных случаев (`POST`, `PUT`, `DELETE`).

Атрибуты `[WebGet]` и `[WebInvoke]` позволяют также использовать шаблон URI с постоянными и переменными частями. Кроме того, эти два атрибута позволяют определять формат сериализации (XML или JSON).

Атрибут `automaticFormatSelectionEnabled` использует заголовок Accept Header входящего сообщения, чтобы автоматически выбрать формат сериализации XML или JSON.

В листинге 4.10 приведен пример использования атрибутов [WebGet] и [WebInvoke] (с учетом базового адреса http://localhost:1234).

Листинг 4.10. Реализация архитектуры REST


```
namespace Wrox
{
    public class Car
    {
        public string Name { get; set; }
        public string Make { get; set; }
        public string Type { get; set; }
        public int Seats   { get; set; }
    }

    [System.Runtime.Serialization.CollectionDataContract(Name="Cars")]
    public class CarPool: List<Car>
    {
        private CarPool()
        {
            this.Add(new Car() { Name = "Dakota",
                Make = "Dodge", Type = "Pickup Truck", Seats = 8 });
            this.Add(new Car() { Name = "TT",
                Make = "Audi", Type = "Sport Car", Seats = 2 });
        }
        private static CarPool AllCars = null;

        public static CarPool GetCarPark()
        {
            if (AllCars ==null)
                AllCars = new CarPool();

            return AllCars;
        }
    }

    [ServiceContract()]
    public interface ICarRentalService
    {
        [OperationContract]
        string HelloWorld();

        [OperationContract]
        [WebGet(UriTemplate = "/CarPool")]
        CarPool GetAllCars();

        [OperationContract]
        [WebGet(UriTemplate = "/CarPool/{carName}")]
        Car GetCar(string carName);

        [OperationContract]
        [WebGet(UriTemplate="/CarPool/{carName}?format=xml",
        ResponseFormat = WebMessageFormat.Xml)]
        Car GetCarXML(string carName);
```

```
    [OperationContract]
    [WebGet(UriTemplate = "/CarPool/{carName}?format=json",
    ResponseFormat = WebMessageFormat.Json)]
    Car GetCarJSON(string carName);

    [OperationContract]
    [WebInvoke(UriTemplate = "CarPool/{carName}", Method = "PUT")]
    Car AddCar(string carName, Car car);

    [OperationContract]
    [WebInvoke(UriTemplate = "/CarPool/{carName}",
    Method = "DELETE")]
    void DeleteCar(string carName);
}

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class CarRentalService: ICarRentalService
{
    public string HelloWorld()
    {
        WebOperationContext.Current.OutgoingResponse.StatusCode =
            System.Net.HttpStatusCode.OK ;
        return "Hello World";
    }
    public CarPool GetAllCars()
    {
        return CarPool.GetCarPark();
    }
    public Car GetCar(string carName)
    {
        return CarPool.GetCarPark().Find(e => e.Name == carName);
    }
    public Car GetCarXML(string carName)
    {
        return GetCar(carName);
    }
    public Car GetCarJSON(string carName)
    {
        return GetCar(carName);
    }
    public void DeleteCar(string carName)
    {
        Car found = CarPool.GetCarPark().Find
            (e => e.Name == carName);
        if (found == null)
            WebOperationContext.Current.OutgoingResponse
                .SetStatusAsNotFound();
        else
            CarPool.GetCarPark().Remove(found);
    }

    public Car AddCar(string carName, Car car)
    {
```

```
            WebOperationContext.Current.OutgoingResponse
            .SetStatusAsCreated
            (
                new Uri("http://localhost:1234/CarPool/" + car.Name)
            );
            CarPool.GetCarPark().Add(car);
            return car;
        }
    }
}
```

В следующем примере команда протокола HTTP `GET URL` является фиксированным текстом и не содержит элементов переменных.

```
[OperationContract]
[WebGet(UriTemplate = "/CarPool")]
CarPool GetAllCars();
```

Ниже показано, как может быть вызвана операция `GetAllCards` (при передаче по адресу команды протокола HTTP `GET`).

```
GET http://loalhost:1234/CarPool
[OperationContract]
[WebGet(UriTemplate = "/CarPool/{carName}")]
Car GetCar(string carName);
```

При вызове следующей строки Leon передается как переменная `{carName}` операции `GetCar` (при передаче команды протокола HTTP `GET` по следующему адресу).

```
GET http://localhost:1234/CarPool/Leon
```

Следующий пример подобен предыдущему примеру. Переменная `{carName}` записывается в переменную `carName`. Передается также фиксированная строка запроса и устанавливается формат ответа `JSON`.

```
[OperationContract]
[WebGet(UriTemplate = "/CarPool/{carName}?format=json", ResponseFormat =
WebMessageFormat.Json)]
Car GetCarJSON(string carName);
```

Как демонстрирует следующий пример, в отличие от атрибута `WebGet` команда может быть определена в атрибуте `WebInvoke`. Переменная `carName` берется из URL, а содержимое объекта `car` из тела запроса.

```
[OperationContract]
[WebInvoke(UriTemplate = "CarPool/{carName}", Method = "PUT")]
Car AddCar(string carName, Car car);
```

Ранее вы видели, как новая привязка `webHttpBinding` совместно с режимом `webHttp` и атрибутом `[WebGet]` или `[WebInvoke]` используется для создания приложения RESTful. Адресуемые ресурсы определяются с помощью шаблонов URI, и вы можете использовать свойство `RequestFormat` или `ResponseFormat`, чтобы решить, какой формат использовать, XML или JSON. При использовании команд протокола HTTP интерфейс API становится интуитивно понятным.

Чтобы определить, была ли операция успешна или нет, используйте коды состояния HTTP и заголовки HTTP, например, доступ к ресурсу может быть ограничен и

требовать передачи имени пользователя и пароля. Для доступа к специфическим элементам HTTP используется класс `WebOperationContext`.

Если бы вызов был успешен, вернулся бы код состояния 200, тогда как код 403 означал бы невозможность найти ресурс. Для облегчения работы с кодами состояния можно использовать перечисление `System.Net.HttpStatusCode` либо вспомогательные методы класса `OutgoingWebResponseContext`.

Следующая строка кода возвратила бы от службы код состояния 200.

```
WebOperationContext.Current.OutgoingResponse.StatusCode = System.Net.HttpStatusCode.OK ;
```

Следующая строка кода, напротив, возвратила бы код состояния 403 (ресурс не найден).

```
WebOperationContext.Current.OutgoingResponse.SetStatusAsNotFound();
```

Работа с клиентами REST

В отличие от платформы WCF с протоколом SOAP архитектура REST не предоставляет никаких средств создания документа WSDL. Поэтому самый быстрый и простой вариант, окно Add Service Reference (Добавление ссылки на службу), не доступен.

Но в этом нет большой беды, а есть даже определенные преимущества. Поскольку использование интерфейса API REST ограничено лишь несколькими командами и лишено обмена сложными сообщениями SOAP, относительно просто создать отдельный прокси-сервер с подходящим стеком, который поддерживает команды протокола HTTP и коды состояния.

Для этого, например, вы могли использовать классы `System.Net.WebRequest` и `System.Net.WebResponse` или задействовать класс `WebChannelFactory` из библиотеки `System.ServiceModel.Web.dll`. В данном случае вы также имеете неограниченный контроль над связью между сервером и клиентом и не должны волноваться ни о каких скрытых действиях и функциональных возможностях создаваемого прокси-сервера.

В отличие от протокола SOAP взаимодействие в стиле REST осуществляется на более высоком уровне, поскольку теоретически клиенту достаточно иметь команду HTTP и протокол XML или JSON. Поэтому, чтобы общаться со службой REST, нет никакой необходимости в обширном стеке SOAP с поддержкой всех протоколов WS*.

Использование комплекта REST Starter Kit

Поскольку архитектура REST доступна только на платформе .NET Framework 3.5 и ей все еще не хватает поддержки в некоторых областях (обработка ошибок, хостинг, клиент, кэширование), вы можете обратиться к использованию комплекта REST Starter Kit. Комплект REST Starter Kit, доступный для загрузки на CodePlex, предоставляет широкий диапазон классов и методов расширения для программирования с использованием стиля REST как на стороне сервера, так и на стороне клиента.

К серверным функциональным возможностям, находящимся в сборке `Microsoft.ServiceModel.Web.dll`, относятся, например, упрощение работы с кодами состояния, интуитивно понятная обработка и передача сообщений об ошибках, ориентированная на ASP.NET поддержка кэширования и отдельный хост службы, позволяющий избежать создания страниц документации вручную. Страница справки создается как документ WSDL на основании кода вашей службы и допускает дальнейшую адаптацию с помощью атрибута `[WebHelp]`.

Класс `HttpClient`, безусловно, — важнейший на стороне клиента. Он упрощает вызов операций REST и поддерживает команды `GET`, `PUT`, `DELETE` и `POST`. Все клиентские функции находятся в сборке `Microsoft.Http.dll` и библиотеке `Microsoft.Http.Extensions.dll`.

Листинг 4.11 иллюстрирует использование класса `HttpClient` для передачи запроса `GET` на чтение данных обо всех доступных автомобилях. Кроме того, в этом примере демонстрируется также использование метода `PUT`.

Листинг 4.11. Клиент REST — класс `HttpClient`


```
namespace Wrox.RestClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (HttpClient restClient = new HttpClient())
            {
                HttpResponseMessage resp =
                restClient.Get("http://localhost:1234/CarPool");
                    resp.EnsureStatusIsSuccessful();
                Console.WriteLine(resp.Headers.ToString());

                var result = resp.Content.ReadAsString();

                Console.WriteLine(result);

                string newCar = "<Car
                xmlns=\"http://schemas.datacontract.org/2004/07/Wrox\">
                <Make>Seat</Make><Name>Leon</Name>
                <Seats>5</Seats><Type>Sport Car</Type></Car>";

                restClient.Put("http://localhost:1234/CarPool/Leon",
                    "application/xml",
                    HttpContent.Create(newCar));
                Console.WriteLine(resp.StatusCode);
            }
        }
    }
}
```

Таким образом, комплект REST Starter Kit предоставляет обширную коллекцию классов и методов, на которую стоит обратить внимание. Кстати, комплект Starter Kit содержит не только сборки, но и весь код, из которого вы можете извлечь множество идей по реализации собственных приложений REST.

Кроме комплекта REST Starter Kit, платформа WCF 4.0 предоставляет дополнительные средства для облегчения программирования для архитектуры REST, такие как новые шаблоны среды разработки Visual Studio, поддержку кэширования для ASP.NET, автоматическое создание страниц справки, класс `WebFaultException`, механизм JSONP, поддержку условного выполнения запроса `GET` и заголовок ETag.

Модель AJAX и платформа WCF

AJAX — это популярная модель программирования, используемая совместно с веб-сайтами, чтобы избежать повторной загрузки всей страницы при необходимости изменить лишь ее часть. Задача заключается в предоставлении сложной интерактивной страницы, не имеющей раздражающих задержек во время отображения. Это означает, что значительно сужается пропасть между простыми веб-сайтами и автономными приложениями рабочего стола.

В основе концепции AJAX лежит тот факт, что браузер или код на стороне клиента посылает асинхронный вызов на сервер, чтобы получить дальнейшую информацию и обновить содержимое страницы возвратившимися данными.

Ранее аббревиатура AJAX означала асинхронный JavaScript и XML (Asynchronous JavaScript and XML), поскольку как язык программирования главным образом использовался язык JavaScript, а для обменена информацией между браузером и сервером использовались документы XML. Однако в последнее время они распространены значительно меньше; сейчас приложение AJAX подразумевает асинхронный вызов с клиентской стороны, который обрабатывается далее с учетом полученных данных.

Как уже упоминалось, язык JavaScript все еще достаточно распространен, но формат JSON намного лучше удовлетворяет требованиям к формату обмена данными, чем формат XML. Формат JSON несколько уже, чем XML, и десериализация его сообщений осуществляется быстрее большинством языков сценариев, таких как JavaScript или PHP. Например, вы можете обратиться к службе REST с помощью кода JavaScript и класса `XMLHttpRequest`. Возвращенный документ XML нужно будет разобрать бит за битом, используя соответствующие классы и методы, а затем обрабатывать далее.

Но если вы используете платформу WCF и язык ASP.NET, то можете сократить длинный путь через класс `XMLHTTPRequest` и выдать прямую инструкцию службе WCF, чтобы загрузить код JavaScript. Затем это можно использовать как прокси-сервер для связи со службой. Это очень похоже на популярный и удобный способ использования автоматически создаваемых прокси-серверов на основе файлов WSDL для приложений .NET.

Чтобы позволить создавать код JavaScript прокси-сервера, замените режим `webHttp` режимом `enableWebScript`. Как показано в листинге 4.12, вы можете получить автоматически созданный исходный код JavaScript через конечную точку при добавлении `/js`.

Листинг 4.12. Прокси-сервер AJAX


```
Type.registerNamespace("wrox.CarRentalService._2009._10");
wrox.CarRentalService._2009._10.RentalService=function() {
wrox.CarRentalService._2009._10.RentalService.initializeBase(this);
this._timeout = 0;
this._userContext = null;
this._succeeded = null;
this._failed = null;
}
wrox.CarRentalService._2009._10.RentalService.prototype={
_get_path:function() {
var p = this.get_path();
if (p) return p;
else return
wrox.CarRentalService._2009._10.RentalService._staticInstance.get_path();},
GetAllCars:function(succeededCallback, failedCallback, userContext) {
return this._invoke(this._get_path(), "GetAllCars",false,{},succeededCallback,
```

```
failedCallback,userContext); },
AddCar:function(car,succeededCallback, failedCallback, userContext) {
return this._invoke(this._get_path(), "AddCar",false,{car:car},succeededCallback,
failedCallback,userContext); }}
...
```

Но вместо того, чтобы вносить изменения в файл конфигурации, можете снова использовать специальный класс `WebScriptServiceHost`, который автоматически настраивает конечную точку на соответствующую привязку и режим.

Если необходимо предоставить службу WCF с помощью создаваемого прокси-сервера JavaScript, то следует помнить, что единственными допустимыми командами будут `GET` и `POST` и что использование шаблонов URI не поддерживается.

Чтобы получить доступ к создаваемому прокси-серверу со страницы ASPX, вы обычно используете элемент управления `ScriptManager` и добавляете адрес конечной точки службы в разделе `ServiceReference`, как показано в листинге 4.13.

Листинг 4.13. Элемент управления `ScriptManager`


```
<asp:ScriptManager ID="ScriptManager1" runat="server">
    <Services>
        <asp:ServiceReference Path="http://hansAt7:1234" />
    </Services>
</asp:ScriptManager>
```

В результате при загрузке страницы сначала с сервера загружается код JavaScript прокси-сервера и используется затем для связи.

Листинги 4.14–4.16 демонстрируют необходимый для этого код на стороне сервера, а также содержимое страницы ASPX. Обратите внимание на то, что для доступа прокси-сервера используется класс `wrox.CarRentalService._2009._10.RentalService`, произведенный как `namespace + servicename`.

Листинг 4.14. Контракт службы REST


```
namespace Wrox.RestClient
{
    [ServiceContract(
        Namespace = "http://wrox/CarRentalService/2009/10",
        Name = "RentalService")]
        public interface ICarRentalService
        {
            [OperationContract]
            CarPark GetAllCars(),

            [OperationContract]
            [WebInvoke(Method = "POST",
            RequestFormat = WebMessageFormat.Xml,
            ResponseFormat = WebMessageFormat.Xml)]
            Car AddCar(Car car),
        }
```

Листинг 4.15. Файл конфигурации для AJAX


```
<services>
    <service name="Wrox.CarRentalService" >
```

```
        <endpoint
            address="http://hansat7:1234/"
            contract="Wrox.ICarRentalService"
            binding="webHttpBinding"
            behaviorConfiguration="webScript"/>
    </service>
</services>
<behaviors>
    <endpointBehaviors>
        <behavior name="webScript">
            <enableWebScript/>
        </behavior>
    </endpointBehaviors>
</behaviors>
```

Листинг 4.16. Файл `Default.aspx`


```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master"
AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="WebAjaxClient._Default" %>
<asp:Content ID="HeaderContent" runat="server"
ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server"
ContentPlaceHolderID="MainContent">
    <asp:ScriptManager ID="ScriptManager1" runat="server">
        <Services>
            <asp:ServiceReference Path="http://hansAt7:1234" />
        </Services>
    </asp:ScriptManager>
    <script>
        function ajaxTest() {
            var myProxy =
            new wrox.CarRentalService._2009._10.RentalService();

            var myCar = new Wrox.Car();
            myCar.Make = "Mercedes";
            myCar.Name = "SLK";
            myProxy.AddCar(myCar, onSuccessAdd, onFailAdd, myProxy);
            myProxy.GetAllCars(onSuccessGetAll, onFailGetAll, myProxy);
        }
        function onSuccessAdd(result) {
            Sget("divAddCar").innerText = result;
        }

        function onFailAdd(result) {
            alert(divAddCar);
        }

        function onSuccessGetAll(result) {
            var allCars = "";

            for (var i = 0; i < result.length; i++) {
                allCars += " " + result[i].Name + " " + result[i].Make;
```

```
            }

            $get("divAllCars").innerText += allCars;
        }
        function onFailGetAll(result) {
            alert(result);
        }

    </script>
    <h2>
        Test for the CarRentalService REST API
    </h2>
    <p>
        WCF Service
        <input type="button" value="Test" onclick="ajaxTest()" />
    </p>
    Add Cars
    <div id="divAddCar">
    </div>
    <p>
        All Cars
        <div id="divAllCars">
        </div>
    </p>
</asp:Content>
```

Результат представлен на рис. 4.4.

Рис. 4.4. Результат выполнения кода

Платформа WCF 4.0 и модуль Silverlight

То, что срабатывает для классических клиентов .NET и клиентов AJAX, работает также на Silverlight. Есть простой способ использования существующей службы WCF, но только с применением окна Add Service Reference (Добавление ссылки на службу) для создания прокси-сервера и программы для него. Если у вас уже есть веб-контейнер Silverlight, используйте новый шаблон элемента Visual Studio под названием Silverlight-enabled WCF Service, чтобы добавить в свой проект файл `.svc` и связанный с ним код (рис. 4.5). (Такие проблемы, как защита и проникновение, здесь не обсуждаются.)

В листинге 4.17 показано содержимое файла `CarRentalService.svc.cs` с проверочным методом `HelloWorld`.

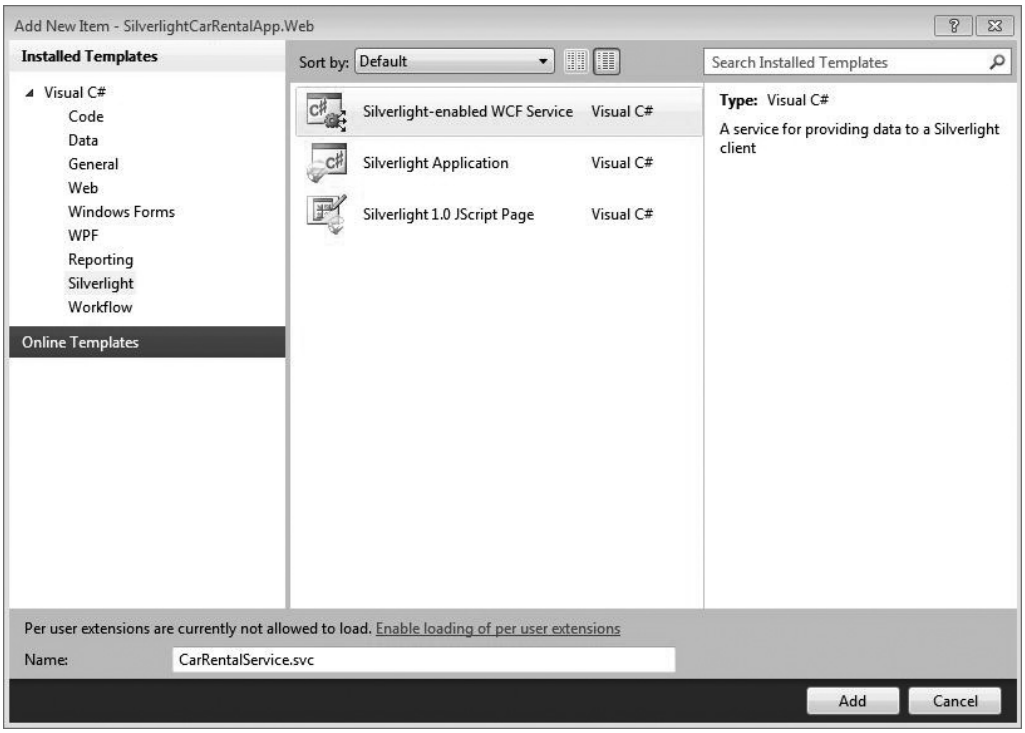


Рис. 4.5. Шаблон Silverlight-enabled WCF Service

Листинг 4.17. Метод `HelloWorld` для Silverlight


```
namespace SilverlightCarRentalApp.Web
{
[ServiceContract(Namespace = "http://WCFBook/SilverlightSamples")]
    [AspNetCompatibilityRequirements(
    RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
    public class CarRentalService
    {
        [OperationContract]
        public string HelloWorld()
        {
            return "Hello Silverlight";
        }
    }
}
```

Чтобы создать прокси-сервер для приложения Silverlight, достаточно добавить в проект ссылку на службу. Листинг 4.18 демонстрирует разметку, а листинг 4.19 — использование созданного прокси-сервера в приложении Silverlight. Обратите внимание на то, что прокси-сервер предлагает только асинхронные вызовы, чтобы предотвратить взаимоблокировки в основном потоке.

Листинг 4.18. Файл MainPage.xaml


```
<StackPanel>
    <Button Content="Button" Height="23"
    Name="button1" Width="75" Click="button1_Click" />
    <TextBox Height="23" Name="textBox1" Width="120" />
</StackPanel>
```

Листинг 4.19. Прокси-сервер WCF Silverlight


```
namespace SilverlightCarRentalApp
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {
            CarRentalProxy.CarRentalServiceClient myClient =
            new CarRentalProxy.CarRentalServiceClient();
            myClient.HelloWorldCompleted += new
            EventHandler<CarRentalProxy.HelloWorldCompletedEventArgs>
            (myClient_HelloWorldCompleted);
            myClient.HelloWorldAsync();
        }

        void myClient_HelloWorldCompleted(object sender,
            CarRentalProxy.HelloWorldCompletedEventArgs e)
        {
            textBox1.Text = e.Result;
        }
    }
}
```

5

Создание экземпляра

В ЭТОЙ ГЛАВЕ...

- Использование различных режимов свойства `InstanceContextMode`
- Что такое сеанс
- Определение свойства `ConcurrencyMode`
- Контроль срока службы
- Использование регулирования
- Полезные советы
- Реализация регулирования

Ваш объект прокси-сервера на стороне клиента (обычно производный от класса `ClientBase<T>`) общается с объектами на стороне сервера через так называемый *стек канала* (channel stack). Хост службы настраивает эти объекты на стороне сервера. Как уже упоминалось в предыдущих главах, между прокси-сервером и объектом на стороне сервера происходит обмен сообщениями SOAP. Свойство `InstanceContextMode` определяет, общается ли во время сеанса клиентский прокси-сервер с одним и тем же экземпляром объекта сервера (режим `PerSession`), или каждый раз, когда используется объект сервера, создается его новый экземпляр, а затем удаляется, как только вызов метода заканчивается (режим `PerCall`), или создается только один экземпляр сервера, независимо от количества клиентов (режим `Single`).

В классических объектно-ориентированных приложениях предполагается, что экземпляр класса создается и используется в течение определенного отрезка времени. Если определенные свойства объекта устанавливаются заранее, они остаются до тех пор, пока есть допустимая ссылка на объект и вызванные методы могут работать со значениями, которые установлены заранее. Если вызывается метод `Dispose` и объектная ссылка установлена в `null`, сборщик "мусора" удаляет объект во время следующего

цикла и освобождает ресурсы. На платформе WCF подобный тип управления экземпляром возможен в режиме `PerSession`.

Однако в распределенной среде ситуация совершенно иная — ссылки указывают не на локальные объекты, а на дистанционные объекты, расположенные на сервере или в другом домене приложения. Поэтому, чтобы создать масштабируемые системы, достаточно эффективные с точки зрения использования ресурсов, применяется подход, отличный от классического *локального* подхода программирования. Объекты сервера обычно создаются на короткий период вызова метода и немедленно освобождаются после его завершения. Значения, необходимые для работы метода, либо передаются каждый раз заново, когда происходит вызов метода, либо загружаются из постоянного хранилища, такого как база данных. Следовательно, в среде с балансом нагрузки не имеет значения, на каком именно сервере создаются объекты; масштабирование до тысяч клиентов не представляет никаких проблем (в этом случае узким местом может стать база данных). Удаление ссылок на объекты не должно явно освобождать их — если приложение-клиент аварийно завершит работу, объекты не останутся *сиротами* (orphaned), поскольку они удаляются непосредственно после каждого вызова. На платформе WCF подобный тип управления экземпляром возможен в режиме `PerCall`.

Иногда очень выгодно иметь только один экземпляр объекта сервера, независимо от количества клиентов. Для этого предназначен режим `Single`. Но выбирать этот вариант нужно лишь тогда, когда вы действительно должны работать только с одним центральным ресурсом.

Свойство `InstanCecontextMode`

При создании экземпляра объекта сервера платформа WCF предоставляет три режима: `PerCall`, `PerSession` и `Single`.

Ваш выбор режима создания экземпляра заметен только на стороне сервера и не отражается на документе WSDL. Поскольку клиент не знает, получит ли он при вызове метода тот же экземпляр и помнит ли все ранее установленные значения, или экземпляр каждый раз создается заново, вы должны проявлять осторожность, проектируя свои операции.

Как уже упоминалось, режим создания экземпляра — это деталь реализации, которая относится только к серверу; его контролирует свойство `InstanceContextMode` атрибута `ServiceBehavior`. Таким образом, вы применяете этот атрибут в своей реализации, а не в контракте службы.

Листинг 5.1 демонстрирует пример применения этого атрибута.

Листинг 5.1. Атрибут `[ServiceBehavior]`

Доступно для загрузки на Wrox.com

```
[ServiceBehavior(
    InstanceContextMode = InstanceContextMode.PerCall)
]
public class CarRentalService: ICarRentalService, IDisposable
{
```

Теперь рассмотрим значения свойства `InstanceContextMode` индивидуально, чтобы, используя различные примеры, подробно изучить их преимущества, недостатки и описать их возможное применение.

Режим PerCall

Прокси-сервер на стороне клиента перенаправляет вызовы метода на сервер. Каждый раз, когда происходит вызов метода, создается новый экземпляр объекта сервера (вызывается стандартный конструктор), а как только работа метода завершается, этот новый экземпляр освобождается. Если объект службы реализует интерфейс `IDisposable`, вызов метода `Dispose` происходит автоматически, после того, как результат посылается прокси-серверу.

Соединение между прокси-сервером и сервером закрывается только тогда, когда вы вызываете метод `Close` своего прокси-сервера.

Объект освобождается после каждого вызова, что делает этот вариант чрезвычайно масштабируемым. Но учтите, что содержимое переменных ваших экземпляров теряется после каждого вызова и его необходимо передавать заново каждый раз или буферизовать данные, а затем снова загружать их при каждом вызове метода. Кроме выгод масштабируемости, нет также никакой необходимости заботиться о проблемах с потоками, поскольку каждый вызов использует отдельный экземпляр.

В листинге 5.2 показаны два пути, которыми клиент может снабдить объект сервера необходимыми данными.

Листинг 5.2. Контроль состояния в режиме PerCall

Доступно для загрузки на Wrox.com

```
double price = 0;
Guid confirmationID;
CarRentalProxy.RentalServiceClient carRentalClient = new
CarRentalProxy.
RentalServiceClient();

Console.WriteLine("Version 1");
price= carRentalClient.CalculatePriceV1(DateTime.Now,
DateTime.Now.AddDays(5),
"Graz", "Wien");
Console.WriteLine("Price to Wien {0}", price);
confirmationID = carRentalClient.ConfirmV1(DateTime.Now,
DateTime.Now.AddDays(5), "Graz", "Wien", price);
Console.WriteLine("ConfirmationID {0}",confirmationID );

Console.WriteLine("Version 2");
CarRentalProxy.PriceCalculationResponse resp =
carRentalClient.CalculatePriceV2(
DateTime.Now, DateTime.Now.AddDays(5), "Graz", "Wien");
Console.WriteLine("Price to Wien {0}", resp.Price );
confirmationID = carRentalClient.ConfirmV2(resp.RequestID );
Console.WriteLine("ConfirmationID {0}", confirmationID);

carRentalClient.ReportCrash(DateTime.Now,"Hartberg",confirmationID);

carRentalClient.Close();
```

Как можно заметить, метод `CalculatePrice` вызывается первым с такими параметрами, как `PickupDate` и `ReturnDate`.

В первом варианте возвращается только вычисляемая цена. Если вы затем захотите вызвать метод `Confirm`, все параметры, включая возвращенное значение, придется передать снова.

Во втором варианте наряду с вычисляемой ценой возвращается идентификатор `requestID`. Этот идентификатор запроса можно использовать, например, для выполнения действия `Confirm`. Во втором варианте для метода `Confirm` важно повторно загрузить исходные значения, используя переданный идентификатор `RequestID`; обычно эти значения загружаются из базы данных.

Метод `ReportCrash` работает так же, как второй вариант. В данном случае методу также передается уникальный идентификатор `confirmationID`.

Листинг 5.3 демонстрирует реализацию службы для режима `PerCall`.

Листинг 5.3. Реализация службы


```
using System;
using Wrox.CarRentalService.Contracts;
using System.ServiceModel;

namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode= InstanceContextMode.PerCall )]
    public class CarRentalService: ICarRentalService, IDisposable
    {

        public CarRentalService()
        {
            Console.WriteLine("CarRentalService Constructor");
        }

        public double CalculatePriceV1(DateTime pickupDate,
            DateTime returnDate,
            string pickupLocation, string returnLocation)
        {
            double price = 0;
            if (returnLocation.Equals("Wien"))
                price=120;
            else
                price =100;

            return price ;
        }

        public Guid ConfirmV1(DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string returnLocation, double price)
        {
            Guid confirmationNumber = Guid.NewGuid();
            // сохранить подтверждение в базе данных
            return confirmationNumber;
        }

        public PriceCalculationResponse CalculatePriceV2(
            DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string returnLocation)
        {
```

```
            PriceCalculationResponse resp = null;
            resp = new PriceCalculationResponse();
            Guid requestId = Guid.NewGuid();
            resp.RequestID = requestId;

            if (returnLocation.Equals("Wien"))
                resp.Price =120;
            else
                resp.Price = 100;
            // Сохранить запрос в базе данных
            return resp ;
        }

        public Guid ConfirmV2(Guid requestID)
        {
            // загрузить запрос из базы данных
            Guid confirmationNumber = Guid.NewGuid();
            // сохранить подтверждение в базе данных
            return confirmationNumber;
        }

        public void ReportCrash(DateTime date,
            String location, Guid confirmationID)
        {
            // загрузить значения из базы данных
            Console.WriteLine(@"
            Crash reported Date {0} Location {1}
            Confirmation ID {2}",
            date,location,confirmationID );
        }
        public void Dispose()
        {
            Console.WriteLine("CarRentalService disposed...");
        }
    }
}
```

Режим `Single`

Как и следует из названия, здесь есть только один экземпляр, независимо от количества клиентских прокси-серверов. Объект создается только однажды, и его срок службы привязан к объекту класса `ServiceHost`.

Главной проблемой этого варианта является масштабируемость. Стандартным потоковым режимом платформы WCF является `ConcurrencyMode.Single`, т.е. объект службы может быть в любой момент блокирован одним потоком для его исключительного использования при продолжительном вызове метода. Объект освобождается только тогда, когда вся работа метода завершена. Все другие вызовы метода ожидают в очереди.

Чтобы частично облегчить эту проблему, вручную измените стандартный режим на `ConcurrencyMode.Multiple`, хотя так вы возьмете на себя ответственность за синхронизацию потоков.

Код листинга 5.4 демонстрирует применение режима `InstanceContextMode.Single`. Установка стандартного режима для свойства `ConcurrencyMode` здесь повторена явно для лучшей наглядности.

Листинг 5.4. Режим `InstanceContextMode.Single`


```
namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
            ConcurrencyMode = ConcurrencyMode.Single )]
    public class CarRentalServiceAT : ICarRentalService, IDisposable
    {
        public int Count { get; set; }
        public CarRentalServiceAT()
        {
            Console.WriteLine("CarRentalService Constructor ");
        }

        public double CalculatePrice(
            DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string returnLocation)
        {
            Count++;
            double returnValue = 0;
            if (returnLocation.Equals("Graz"))
                returnValue = 100;
            else
                returnValue = 150;
            Console.WriteLine("total number of price calculations {0}",
                              Count);
            return returnValue;
        }

        public void Dispose()
        {
            Console.WriteLine("CarRentalService disposed ");
        }
    }
}
```

Листинг 5.5 демонстрирует, что у класса `ServiceHost` есть перегруженный вариант, которому может быть передан готовый объект `Singleton`. Для доступа к объекту `Singleton` может быть также использовано свойство `SingletonInstance` класса `ServiceHost`.

Листинг 5.5. Класс `ServiceHost` и объект `Singleton`


```
using System.ServiceModel;
...

CarRentalServiceAT myService = new CarRentalServiceAT();
myService.Count = 20;
```

```
ServiceHost carRentalHost = null;
carRentalHost = new ServiceHost(myService);

carRentalHost.Open();
CarRentalServiceAT singleton = null;
carRentalHost.SingletonInstance as CarRentalServiceAT;
singleton.Count = 0;
```

Режим PerSession

Режим PerSession означает, что каждый клиентский прокси-сервер общается со своим собственным экземпляром объекта на стороне сервера. Объект на стороне сервера существует, пока клиент не вызовет метод Close прокси-сервера или пока не истечет период ожидания сеанса (значение по умолчанию — 10 минут).

После того как подключение было завершено явно, при вызове метода Close или при удалении объекта сервера по истечении периода ожидания, прокси-сервер больше не может использоваться. При попытке вызова метода вы получите исключение класса CommunicationException.

Преимущество создания экземпляра PerSession заключается в том, что установленные значения сохраняются на стороне сервера без дополнительных усилий. Но масштабируемость при этом страдает, поскольку каждое активное подключение, естественно, занимает объем памяти на сервере, независимо от того, требует ли клиент в настоящее время этого или нет. При аварийном отказе клиента или если не вызван метод Close, значения остаются в памяти до конца сеанса. Код листинга 5.6 демонстрирует использование режима InstanceContextMode.PerSession.

Листинг 5.6. Режим InstanceContextMode.PerSession


```
using System;
using Wrox.CarRentalService.Contracts;
using System.Diagnostics;
using System.ServiceModel;

namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode= InstanceContextMode.PerSession)]
    public class CarRentalService: ICarRentalService, IDisposable
    {
        private DateTime pickupDate, returnDate;
        private String pickupLocation, returnLocation;
        private string confirmationNumber;
        public CarRentalService()
        {
            Console.WriteLine("CarRentalService Constructor");
        }
        public void SetValues(DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string returnLocation)
        {
            this.pickupDate = pickupDate;
            this.returnDate = returnDate;
            this.pickupLocation = pickupLocation;
            this.returnLocation = returnLocation;
        }
```

```
        public double CalculatePrice()
        {
            double price = 0;
            if (returnLocation.Equals("Wien"))
                price=120;
            else
                price =100;

            return price ;
        }

        public string Confirm()
        {
            confirmationNumber = OperationContext.Current.SessionId ;

            return confirmationNumber;
        }

        public void Dispose()
        {
            Console.WriteLine("CarRentalService disposed...");
        }
    }
}
```

Как видно в листинге 5.7, значения устанавливаются при первом вызове, и закрытые переменные экземпляра применяются при последующих вызовах без повторной передачи с клиента на сервер. Идентификатор подтверждения соответствует идентификатору `SessionId`.

Листинг 5.7. Клиент `PerSession`


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            double price = 0;
            CarRentalProxy.RentalServiceClient carRentalClient = null;
            carRentalClient = CarRentalProxy.RentalServiceClient();
            carRentalClient.SetValues(
                DateTime.Now, DateTime.Now.AddDays(5),
                "Graz", "Wien");
            price = carRentalClient.CalculatePrice();
            Console.WriteLine("Price to Wien {0}", price);

            carRentalClient.SetValues(
                DateTime.Now, DateTime.Now.AddDays(5),
```

```
                "Graz", "Villach");
            price = carRentalClient.CalculatePrice();
            Console.WriteLine("Price to Villach {0}", price);

            string confirmNumber = carRentalClient.Confirm();
            Console.WriteLine("Reservation ID {0}", confirmNumber);
            carRentalClient.Close();
            try
            {
                // заказ уже подтвержден, и прокси-сервер закрт
                carRentalClient.SetValues(DateTime.Now,
                DateTime.Now.AddDays(5), "Graz", "Salzburg");
                price = carRentalClient.CalculatePrice();
                Console.WriteLine("Price to Salzburg {0}", price);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message );
            }
        }
    }
}
```

Режим `PerSession` требует использования протокола, поддерживающего сеанс. Например, можете использовать привязку `netTcpBinding` с ориентированным на соединения протоколом передачи TCP, позволяющим серверу перенаправлять входящие сообщения на правильный серверный объект. Привязка `basicHttpBinding`, например, не подходит, поскольку протокол передачи HTTP не требует установки соединения и не может перенаправлять входящие сообщения на конкретный экземпляр сервера.

Но есть некоторые привязки, которые используют протокол передачи HTTP и оказывают дополнительную поддержку с точки зрения защиты или надежности. В данном случае платформа WCF использует идентификатор такого протокола для установки индивидуального подключения между прокси-сервером и объектом сервера.

В листинге 5.8 показано использование привязки `wsHttpBinding`, которая обеспечивает и защиту, и надежность. Если вы теперь активизируете защиту, например, то платформа WCF использует идентификатор безопасности для сопоставления клиента и сервера, а также для идентификатора `SessionId` (рис. 5.1).

Листинг 5.8. Привязка `wsHttpBinding`


```
<bindings>
    <wsHttpBinding>
        <binding name="wsReliable" >
            <reliableSession enabled="false"/>
            <security mode="Message"/>
        </binding>
    </wsHttpBinding>
</bindings>
<service
behaviorConfiguration="MetaDataBehavior" name="Wrox.CarRentalService.
Implementations.Europe.CarRentalService">
<endpoint
    address=""
```

```
    binding="wsHttpBinding" bindingConfiguration="wsReliable"
    contract="Wrox.CarRentalService.Contracts.ICarRentalService"
/>
</service>
```

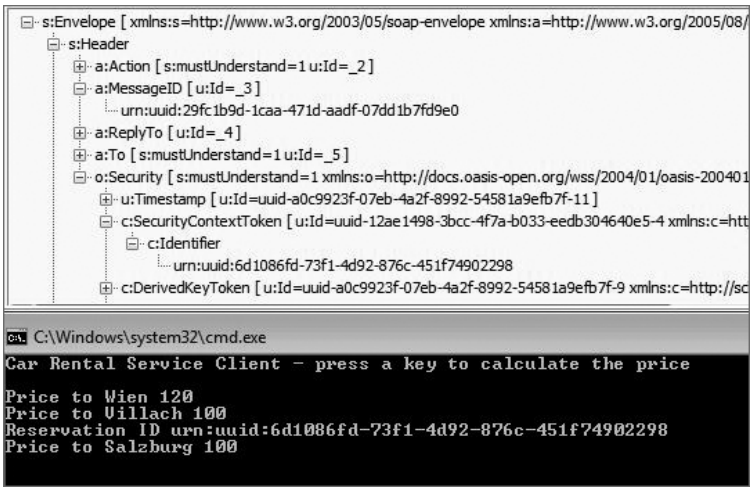


Рис. 5.1. Использование идентификатора

Учитывая тот факт, что комбинация режима `InstanceContextMode.PerSession` и привязки `basicHttpBinding` не приводит к ошибке во время выполнения программы, но и не обеспечивает ожидаемого поведения (поскольку экземпляр сервера ведет себя так, как в режиме `PerCall`), необходимо запросить или отклонить протокол, который поддерживает сеансы явно. Для этого в контракте службы применяется свойство `SessionMode` (листинг 5.9).

Листинг 5.9. Применение свойства `SessionMode` для установки подходящей привязки

Доступно для загрузки на Wrox.com

```
 [ServiceContract(
    Namespace = " http://wrox/CarRentalService/2009/10",
    SessionMode = SessionMode.Required ,
    Name = "RentalService")]
public interface ICarRentalService...
```

Если попытаться загрузить эту службу с привязкой `basicHttpBinding`, во время выполнения программы, когда определение службы читается хостом службы, происходит следующая ошибка.

```
Unhandled Exception: System.InvalidOperationException: Contract
requires Session, but Binding 'BasicHttpBinding' doesn't support it
or isn't configured properly to support it.
```

(Необработанное исключение: `System.InvalidOperationException`: Контракт требует сеанса, но привязка `'BasicHttpBinding'` не поддерживает его или не была настроена правильно, чтобы поддержать его.)

С другой стороны, комбинация значения `SessionMode.NotAllowed` и привязки `netTcpBinding` приводит к следующему сообщению об ошибке.

```
Unhandled Exception: System.InvalidOperationException: Contract does
not allow Session, but Binding 'NetTcpBinding' does not support
Datagram or is not configured properly to support it.
```

(Необработанное исключение: `System.InvalidOperationException`: Контракт не допускает сеансов, но привязка `'NetTcpBinding'` не поддерживает дейтаграммы или не настроена правильно, чтобы поддерживать их.)

Независимо от предыдущих конфигураций, в режиме `PerSession` свойство `Reliability` всегда должно оставаться активизированным.

Срок службы

Как уже упоминалось, вы можете освободить экземпляр сервера явно при вызове метода `Close` или по истечении стандартного периода ожидания. Но при использовании сеансов нередко важно вызывать методы в определенном порядке. Например, сначала следует вызвать метод `SetValues` и только затем вычислять цену. Кроме того, когда вы вызываете метод `Confirm`, контракт между клиентом и компанией проката автомобилей должен быть заключен, а затем объект сервера может быть удален.

Чтобы определить порядок вызова методов, используются свойства `IsInitiating` и `IsTerminating`. Свойство `IsInitiating` означает, что этот метод может запустить сеанс или использоваться в существующем сеансе. Свойство `IsTerminating` означает, что сеанс заканчивается, если этот метод вызван; где необходимо, вызывается метод `Dispose` для объекта сервера, и прокси-сервер становится неприменимым. Установка данных свойств затрагивает также созданный документ WSDL (листинг 5.10), а, следовательно, это может быть также замечено и использовано клиентом.

Листинг 5.10. Свойства `IsInitiating` и `IsTerminating`, а также документ WSDL


```
<wsdl:operation
    msc:isInitiating="false"
    msc:isTerminating="false"
    name="CalculatePrice"
>
```

Если вы, например, определяете контракт операции, как представлено в листинге 5.11, то метод не может быть вызван первым. При попытке клиента вызвать этот метод, последует сообщение об ошибке.

```
Unhandled Exception: System.InvalidOperationException: The operation
'CalculatePrice' cannot be the first operation to be called because
IsInitiating is false.
```

(Необработанное исключение: `System.InvalidOperationException`: Операция `'CalculatePrice'` не может быть первой вызванной операцией, поскольку `IsInitiating` содержит значение `false`.)

Листинг 5.11. Свойства `IsInitiating` и `IsTerminating`


```
[OperationContract(IsInitiating = false , IsTerminating = false)]
double CalculatePrice();
```

Таким образом, окончательный контракт службы для примера проката автомобилей показан в листинге 5.12.

Листинг 5.12. Контракт службы проката автомобилей


```
[ServiceContract(
    Namespace = "http://wrox/CarRentalService/2009/10"
    SessionMode = SessionMode.Required ,
    Name = "RentalService")]
public interface ICarRentalService
{
    [OperationContract(IsInitiating=true, IsTerminating =false)]
        void SetValues(DateTime pickupDate, DateTime returnDate,
        string pickupLocation, string returnLocation);

    [OperationContract(IsInitiating = false , IsTerminating = false)]
    double CalculatePrice();

    [OperationContract(IsInitiating = false , IsTerminating = true )]
    string Confirm();
}
```

Следующая дополнительная возможность повлиять на срок службы экземпляра службы заключается в использовании атрибута `OperationBehavior` совместно со свойством `ReleaseInstanceMode`. В принципе каждый экземпляр сервера окружается контекстом, который поддерживает фактическое соединение с клиентским прокси-сервером. В определенных случаях может быть выгодно поддерживать контекст (контейнер для экземпляра), повторно создавать содержавшийся в нем экземпляр перед вызовом метода и удалять его после завершения работы метода. Фактически свойство `ReleaseInstanceMode` используется только совместно с режимом `PerSession`, если пользователь хочет сохранить сеанс, но не экземпляр.

В листинге 5.13 методу `SetValues` был присвоен режим `ReleaseInstanceMode.BeforeCall`. В результате существующий экземпляр удаляется и создается снова перед каждым вызовом этого метода.

Листинг 5.13. Режим `ReleaseInstanceMode.BeforeCall`


```
[OperationBehavior(ReleaseInstanceMode=ReleaseInstanceMode.BeforeCall)]
public void SetValues(DateTime pickupDate, DateTime returnDate,
                      string pickupLocation, string returnLocation)
```

Кроме того, в контракт службы был включен следующий метод, `Reset` (листинг 5.14), и был присвоен режим `ReleaseInstanceMode.AfterCall`. Теперь экземпляр удаляется каждый раз, когда клиент вызывает метод `Reset`.

Листинг 5.14. Режим `ReleaseInstanceMode.AfterCall`


```
[OperationBehavior(ReleaseInstanceMode=ReleaseInstanceMode.AfterCall )]
public void Reset()
```

Другими возможностями были бы `ReleaseInstanceMode.BeforeCall` (значение по умолчанию) и `ReleaseInstanceMode.BeforeAndAfterCall`.

Эти возможности управления экземпляром должны использоваться только в специальных, исключительных случаях, если необходим непосредственный контроль над продолжительностью существования экземпляра сервера по причинам производительности или масштабируемости.

Идентификатор SessionId

Идентификатор SessionId, в форме GUID, передается между клиентом и сервером, чтобы уникально идентифицировать соединение между клиентом (прокси-сервер) и сервером (контекст). Он происходит либо из сеанса протокола передачи (TCP), либо из логического идентификатора сеанса, если вы используете привязку wsHttpBinding. Идентификатор SessionId используется прежде всего для регистрации и протокола и может применяться независимо от режима PerCall, PerSession или Single.

Поэтому идентификатор SessionID доступен всегда, когда вы используете протокол или привязку, поддерживающую сеанс. Например, можно использовать протокол TCP, который так или иначе требует установки соединения, или привязку wsHttpBinding, которая использует не требующий установки соединения протокол передачи HTTP, но несет поверх него логический идентификатор сеанса.

В первую очередь идентификатор сеанса используется тогда, когда вы активизировали привязку wsHttpBinding, чтобы обеспечить безопасность и надежность. В этом случае как идентификатор сеанса используется идентификатор безопасности. Соединение между идентификатором сеанса и идентификатором безопасности также показано на рис. 5.1.

В любом случае с используемой привязкой важно активизировать сеансы защиты или надежности.

Следовательно, идентификатор сеанса доступен всегда, когда задействуется надежность и/или защита. В дополнение к этим ограничениям для связи между прокси-сервером и сервером также важен обмен идентификатором. Для этого прокси-сервер может быть открыт вручную, что рекомендуется, или, по крайней мере, при вызове одного из методов. Если вы будете обращаться к идентификатору SessionId до связи или использовать протокол, который не поддерживает сеансов, то получите либо неправильный идентификатор, либо сообщение об ошибке.

Обратиться к идентификатору SessionId на клиентской стороне можно с помощью свойства прокси-сервера InnerChannel. Этот идентификатор предоставляется на стороне сервера атрибутом Current контекста операции.

Учтите также то, что если вы активизировали режим SessionMode.NotAllowed на стороне сервера, то можете обратиться к идентификатору SessionId, даже если он пуст.

Код листинга 5.15 приводит к выводу на консоль, представленному в листинге 5.16.

Листинг 5.15. Идентификатор SessionId на стороне клиента

Доступно для загрузки на Wrox.com

```
namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            CarRentalProxy.RentalServiceClient carRentalClient = null;
            carRentalClient = new CarRentalProxy.RentalServiceClient();

            double price = 0;

            try
            {
                Console.WriteLine(
                    carRentalClient.InnerChannel.SessionId);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message ); ;
            }

            carRentalClient.Open();
            price = carRentalClient.CalculatePrice(
                DateTime.Now, DateTime.Now.AddDays(5), "Graz", "Wien");
            Console.WriteLine("Price {0}", price);
            Console.WriteLine(carRentalClient.InnerChannel.SessionId);

            carRentalClient.Close();
            try
            {
                Console.WriteLine(
                carRentalClient.InnerChannel.SessionId);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message); ;
            }

            carRentalClient = new CarRentalProxy.RentalServiceClient();

            price = carRentalClient.CalculatePrice(
                DateTime.Now,DateTime.Now.AddDays(5),"Graz","Villach");
            Console.WriteLine("Price {0}", price);

            Console.WriteLine(carRentalClient.InnerChannel.SessionId);

            price = carRentalClient.CalculatePrice(
                DateTime.Now,DateTime.Now.AddDays(5),"Graz", "Villach");
            Console.WriteLine("Price {0}", price);

            Console.WriteLine(carRentalClient.InnerChannel.SessionId);
            Console.WriteLine("try it again ... demo for Single");
            Console.ReadLine();
            price = carRentalClient.CalculatePrice(
                DateTime.Now,DateTime.Now.AddDays(5),"Graz","Villach");
            Console.WriteLine("Price {0}", price);

            Console.WriteLine(carRentalClient.InnerChannel.SessionId);
        }
    }
}
```

Листинг 5.16. Вывод клиента


```
The session channel must be opened before the session ID can be accessed.
Price 120
urn:uuid:9dd6c7de-d89a-4200-9be4-59f9e5dfc0bc
urn:uuid:9dd6c7de-d89a-4200-9be4-59f9e5dfc0bc
Price 120
urn:uuid:c3 606853-13a0-4fc0-aa4e-f859a2d0891a
Price 120
urn:uuid:c3606853-13a0-4fc0-aa4e-f859a2d0891a
try it again ... demo for Single

Price 120
urn:uuid:c3606853-13a0-4fc0-aa4e-f859a2d0891a
```

К идентификатору `SessionId` можно обратиться через свойство `InnerChannel` прокси-сервера. Чтобы получить доступ к идентификатору `SessionId`, следует открыть соединение между прокси-сервером и сервером; это может быть сделано либо явно, с помощью метода `Open`, либо при вызове по крайней мере одного метода. После того как прокси-сервер будет закрыт, доступным становится старый идентификатор `SessionId`. Когда прокси-сервер открывается снова, о новом идентификаторе `SessionId` договариваются заново.

Серверный код листинга 5.17 с конфигурацией, представленной в листинге 5.18, приводит к выводу, представленному в листинге 5.19.

Листинг 5.17. Идентификатор `SessionId` на стороне сервера


```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single )]
    public class CarRentalService: ICarRentalService, IDisposable
    {
        public CarRentalService()
        {
            Console.WriteLine("CarRentalService Constructor" );
        }

        public double CalculatePrice(
            DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string returnLocation)
        {
            OperationContext ctx = OperationContext.Current;
            Console.WriteLine("CalculatePrice \t{0} ", ctx.SessionId);
            if (returnLocation.Equals("Villach"))
                return 50;
            else
                return 120;
        }
        public void Dispose()
        {
            OperationContext ctx = OperationContext.Current;
            Console.WriteLine("Disposed \t{0}", ctx.SessionId);
        }
    }
}
```

Листинг 5.18. Файл конфигурации


```
<system.serviceModel>
    <bindings>
        <wsHttpBinding>
            <binding name="mywsBinding">
                <reliableSession enabled="false"/>
                <security mode="Message"/>
            </binding>
        </wsHttpBinding>
    </bindings>
    <services>
        <service behaviorConfiguration="MetaDataBehavior"
            name="Wrox.CarRentalService.Implementations.
            Europe.CarRentalService">
        <endpoint address=""
            binding="wsHttpBinding"
            bindingConfiguration="mywsBinding"
            contract="Wrox.CarRentalService.Contracts.ICarRentalService"
        />
        </service>
    </services>
</system.serviceModel>
```

Листинг 5.19. Вывод сервера


```
The car rental service is up and running...
CalculatePrice urn:uuid:9dd6c7de-d89a-4200-9be4-59f9e5dfc0bc
CalculatePrice urn:uuid:c3 60 6853-13a0-4fc0-aa4e-f859a2d0891a
CalculatePrice urn:uuid:c3 60 6853-13a0-4fc0-aa4e-f859a2d0891a
CalculatePrice  urn:uuid:c3606853-13a0-4fc0-aa4e-f859a2d0891a
```

К идентификатору `SessionId` можно обратиться через свойство `OperationContext.Current.SessionId`. Решающим фактором для общего идентификатора `SessionId` является не свойство `InstanceContextMode`, а используемая привязка. К идентификатору `SessionId` можно также обратиться в методе `Dispose`, хотя это имеет смысл только в режиме `PerCall` или `PerSession`.

Производительность

Производительность служб WCF зависит от множества факторов. Кроме общих тем, таких как процессор, оперативная память и сетевая производительность, важнейшую роль играют такие специфические для платформы WCF факторы, как конструкции `InstanceContextMode`, `ConcurrencyMode`, `DataContract`, или используемая привязка.

Свойство `InstanceContextMode` используется для контроля режима создания экземпляра объекта службы, возможные варианты: `PerCall`, `PerSession` и `Singleton`.

Привязка определяет используемый протокол передачи и кодировку символов. Кроме того, привязка позволяет применять широкий диапазон протоколов WS*.

Свойство `ConcurrencyMode` описывает, разрешено ли нескольким потокам одновременно получить доступ к одному и тому же объекту. Свойство `ConcurrencyMode` контролирует атрибут `[ServiceBehavior]`, а его значением по умолчанию является

ConcurrencyMode.Single. Альтернативой являются значения ConcurrencyMode.Multiple и ConcurrencyMode.Reentrant. Режим ConcurrencyMode.Single означает, что к объекту сервера может обратиться только один поток. Это позволит избежать проблем синхронизации, поскольку остальные запросы автоматически помещаются в очередь, пока первый поток не освободит объект. Режим ConcurrencyMode.Multiple означает, что к одному и тому же объекту может обращаться любое количество потоков. При необходимости вам придется позаботиться о синхронизации потока вручную с помощью классических средств .NET, таких как классы Monitor или Mutex.

Режим ConcurrencyMode.Reentrant особенно важен для сценариев обратного вызова и не рассматривается здесь подробно.

В принципе свойство ConcurrencyMode играет важную роль только при использовании многопоточного клиента, который обращается к объекту в режиме PerSession или Singleton. В случае режима PerCall для каждого вызова метода в любом случае создается новый объект, и проблемы потоков не возникает изначально.

Использование атрибута [ServiceBehavior] демонстрирует листинг 5.20.

Листинг 5.20. Атрибут [ServiceBehavior]

Доступно для загрузки на Wrox.com

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
    ConcurrencyMode=ConcurrencyMode.Single)]
public class CarRentalService: ICarRentalService, IDisposable
{
```

Регулирование

Платформа WCF предоставляет и другую возможность для увеличения производительности и предотвращения перегрузки сервера, а следовательно, защиты от атак DoS, например. С помощью элемента <serviceThrottling> в разделе ServiceBehavior файла конфигурации вы можете контролировать предельные значения, указанные ниже. Программный пример приведен в листинге 5.21.

- Атрибут maxConcurrentCalls. Максимальное количество операций службы, способных выполняться одновременно. При превышении этого значения последующие вызовы метода помещаются в очередь и обрабатываются впоследствии.
- Атрибут maxConcurrentSessions. Максимальное количество одновременных передач или сеансов приложения.
- Атрибут maxConcurrentInstances. Максимальное количество экземпляров.

Листинг 5.21. Режим serviceThrottling

Доступно для загрузки на Wrox.com

```
<serviceBehaviors>
    <behavior name="throttlingBehavior">
        <serviceThrottling
            maxConcurrentCalls ="5"
            maxConcurrentSessions="2"
```

```
            maxConcurrentInstances="3" />
    </behavior>
</serviceBehaviors>
```

Для чтения значений регулирования можете использовать код, представленный в листинге 5.22.

Листинг 5.22. Чтение значений регулирования


```
namespace Wrox.CarRentalService.ConsoleHost
{
    class Program
    {
        static void Main(string[] args)
        {
            System.ServiceModel.ServiceHost carRentalHost = null;
            carRentalHost = new ServiceHost(
                typeof(Wrox.CarRentalService.Implementations.
                       Europe.CarRentalService));
            carRentalHost.Open();
            ChannelDispatcher dispatcher =
                carRentalHost.ChannelDispatchers[0]
                as ChannelDispatcher;
            Console.WriteLine("Max concurrent calls {0}",
                dispatcher.ServiceThrottle.MaxConcurrentCalls);
            Console.WriteLine("Max concurrent instances {0}",
                dispatcher.ServiceThrottle.MaxConcurrentInstances);
            Console.WriteLine("Max concurrent sessions {0}",
                dispatcher.ServiceThrottle.MaxConcurrentSessions);
            Console.WriteLine(9"The car rental service is up
                and running...");
            Console.ReadLine();
        }
    }
}
```

С помощью этого атрибута `ServiceBehavior` контролируется количество экземпляров или сеансов, способных существовать одновременно, а также разрешенное количество одновременных вызовов. Используемые значения по умолчанию можно изменить в зависимости от вашей среды и достигнуть лучшей общей производительности. Но наилучшие значения регулирования зависят от множества обстоятельств. Вы должны знать ответы на следующие вопросы: какой объем памяти занимает сеанс? Сколько параллельных клиентов ожидается? Многопоточные ли клиенты? Какой режим свойства `InstanceContextMode` используется? Какая привязка используется?

Во всех случаях настройки важно иметь некую базовую линию, а после внесения изменений в конфигурацию сравнить эту базовую линию с новыми значениями, чтобы выяснить, оказали ли эти изменения на самом деле позитивное воздействие, или общая нагрузка на вашем сервере снизилась вообще. Платформа WCF предоставляет множество системных счетчиков для контроля производительности или создания базовой линии. Счетчики производительности следует активизировать в конфигурации или программно прежде, чем читать их значения.

Пример активизации с помощью файла конфигурации приведен в листинге 5.23.

Листинг 5.23. Активизация счетчиков производительности


```
<system.serviceModel>
...
    <diagnostics performanceCounters="All"/>
...
</system.serviceModel>
```

После активизации счетчиков производительности и запуска хоста службы вы сможете контролировать текущие значения производительности с помощью системного монитора Windows; платформа WCF предоставляет индикаторы производительности для службы в целом: для конечных точек и для операций.

Чтобы выяснить, какой эффект производят отдельные компоненты (`serviceThrottling`, `InstanceContextMode`, `ConcurrencyMode`), начните с простой службы в режиме `PerCall` с привязкой `basicHttpBinding` и постепенно перебирайте все варианты конфигурации.

Отправная точка — это существующая операция `CalculatePrice` службы `CarRentalService`. Чтобы сделать эффект от регулирования более заметным, сократите параметры регулирования так, как показано в листинге 5.24.

Листинг 5.24. Регулирование службы


```
<serviceThrottling
    maxConcurrentCalls ="5"
    maxConcurrentInstances="10"
    maxConcurrentSessions="15"
/>
```

Реализация службы представлена в листинге 5.25.

Листинг 5.25. Реализация службы


```
namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
        ConcurrencyMode=ConcurrencyMode.Reentrant)]
    public class CarRentalService: ICarRentalService, IDisposable
    {
        public double CalculatePrice(
            DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string vehiclePreference)
        {
            System.Threading.Thread.Sleep(5000);
            return double.Parse(pickupLocation);
        }
        public double GetDummyNumber()
        {
            return 10;
        }

        public CarRentalService()
        {
            Console.WriteLine("CarRentalService Constructor");
```

```
        }

        public void Dispose()
        {
            Console.WriteLine("CarRentalService disposing...");
            System.Threading.Thread.Sleep(2000);
            Console.WriteLine("CarRentalService disposing...");
        }
    }
}
```

Код клиента, представленный в листинге 5.26, создает с помощью привязки `basicHttpBinding` одиночный прокси-сервер, а затем асинхронно вызывает метод `CalculatePrice` сорок раз.

Листинг 5.26. Код клиента


```
namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            CarRentalProxy.RentalServiceClient carRentalClient = new
CarRentalProxy.RentalServiceClient("NetTcpBinding_RentalService");

            for (int i = 0; i < 40; i++)
            {
                carRentalClient.BeginCalculatePrice
                    (DateTime.Now, DateTime.Now.AddDays(5),
                     i.ToString(),
                     "Graz",
                     priceCalcFinished,
                     carRentalClient);
            }

            Console.ReadLine();
        }
        public static void priceCalcFinished(IAsyncResult asyncResult)
        {
            Console.WriteLine("priceCalcFinished");
            try
            {
                RentalServiceClient carRentalClient =
                    asyncResult.AsyncState as RentalServiceClient;
                double price = carRentalClient.EndCalculatePrice(asyncResult);
                Console.WriteLine("Price {0}", price);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message); ;
            }
        }
    }
}
```

После запуска этого примера вы увидите, что вычисления цены осуществляются в блоках по пять, поскольку в любой момент времени разрешено максимум пять вызовов. Остальные вызовы буферизируются и обрабатываются поэтапно. Установка значения 5 для атрибута `maxConcurrentInstances` привела бы к тому же результату. Изменение атрибута `maxConcurrentSessions` не имеет в этом случае абсолютно никакого значения, поскольку не используется ни протокол передачи, требующий установки соединения, ни привязка `ws` с защитой или надежностью. Когда обработка отдельных вызовов происходит слишком долго, клиент получит исключение периода ожидания; по умолчанию для этого задано значение в одну минуту. Промежуток между вызываемым методом и передачей исключения периода ожидания может быть изменен на клиентской стороне с помощью атрибута `sendTimeout` вашей привязки.

Если сменить привязку на `netTCPBinding`, атрибут `maxConcurrentSessions` не сработает, даже при том, что протокол TCP поддерживает подключения на уровне протокола передачи. Это связано с тем, что на стороне клиента был создан только один прокси-сервер. В результате атрибут `maxConcurrentSessions` не имеет никакого эффекта.

Но если вы переместите создание экземпляра прокси-сервера в цикл, то каждый его экземпляр установит отдельное соединение со стороной сервера.

Листинг 5.27 демонстрирует измененный код клиента.

Листинг 5.27. Измененный код клиента


```
for (int i = 0; i < 40; i++)
{
    CarRentalProxy.RentalServiceClient carRentalClient = new CarRentalProxy.
    RentalServiceClient("NetTcpBinding_RentalService");
    carRentalClient.BeginCalculatePrice
    (
     DateTime.Now, DateTime.Now.AddDays(5),
     i.ToString(),
     "Graz",
     priceCalcFinished,
     carRentalClient);
}
```

На основании конфигурации в листинге 5.28 возможно только пять активных экземпляров. Если значения двух других атрибутов, `maxConcurrentCalls` или `maxConcurrentInstances`, окажутся ниже, то за пределы, конечно, будут приняты они.

Листинг 5.28. Измененный режим `serviceThrottling`


```
<serviceThrottling
maxConcurrentCalls ="100"
maxConcurrentInstances="100"
maxConcurrentSessions="5" />
```

В этом случае для прокси-сервера также чрезвычайно важно быть заключенным в методе обратного вызова, как представлено в листинге 5.29; в противном случае поддержка сеанса продолжится и он будет закрыт только после 10-минутного значения по умолчанию.

Листинг 5.29. Закрытие прокси-сервера в методе обратного вызова


```
public static void priceCalcFinished(IAsyncResult asyncResult)
{
    Console.WriteLine("priceCalcFinished");
    try
    {
        RentalServiceClient carRentalClient =
            asyncResult.AsyncState as RentalServiceClient;
        double price =
            carRentalClient.EndCalculatePrice(asyncResult);
        Console.WriteLine("Price {0}", price);
        carRentalClient.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message); ;
    }
}
```

Заключительный пример — использование режимов `ConcurrencyMode.Multiple` и `InstanceContextMode.PerSession` с многопоточным клиентом. Для этого код сервера изменен таким образом, чтобы значения переменных экземпляра увеличивались в методе `CalculatePrice`, как представлено в листинге 5.30.

Листинг 5.30. Режимы `ConcurrencyMode.Multiple` и `PerSession`


```
namespace Wrox.CarRentalService.Implementations.Europe
{
    [ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession ,
    ConcurrencyMode=ConcurrencyMode.Multiple )]
    public class CarRentalService: ICarRentalService, IDisposable
    {
        private object myLock = new object();
        private int Counter;
        public double CalculatePrice(
            DateTime pickupDate, DateTime returnDate,
            string pickupLocation, string vehiclePreference)
        {
            lock (myLock)
            {
                int tempValue = Counter;
                tempValue++;
                System.Threading.Thread.Sleep(2000);
                Counter = tempValue;
                Console.WriteLine("Counter {0}", Counter);
            }
            return double.Parse(pickupLocation);
        }

        public CarRentalService()
        {
            Console.WriteLine("CarRentalService Constructor");
        }
```

```
        public void Dispose()
        {
            Console.WriteLine("CarRentalService disposing...");
            System.Threading.Thread.Sleep(2000);
            Console.WriteLine("CarRentalService disposing...");
        }
    }
}
```

Если многопоточный клиент вызовет метод `CalculatePrice` теперь, как показано в листинге 5.31, в вычислении произойдет логическая ошибка.

Листинг 5.31. Многопоточный клиент


```
namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            CarRentalProxy.RentalServiceClient carRentalClient = new
CarRentalProxy.RentalServiceClient("NetTcpBinding_RentalService");
            for (int i = 0; i < 40; i++)
            {
                carRentalClient.BeginCalculatePrice(DateTime.Now,
                    DateTime.Now.AddDays(5),
                    i.ToString(),
                    "Graz",
                    priceCalcFinished,
                    carRentalClient );
            }
            Console.ReadLine();
        }
        public static void priceCalcFinished(IAsyncResult asyncResult)
        {
            Console.WriteLine("priceCalcFinished");
            try
            {
                RentalServiceClient carRentalClient =
                    asyncResult.AsyncState as RentalServiceClient;
                double price =
                    carRentalClient.EndCalculatePrice(asyncResult);
                Console.WriteLine("Price {0}", price);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message ); ;
            }
        }
    }
}
```

Эта проблема решается либо за счет переустановки свойства `ConcurrencyMode` в режим `Single`, либо с помощью ручной синхронизации в методе `CalculatePrice`.

Полезные советы

Как вы уже видели в предыдущих примерах, на общую производительность и предел нагрузки вашей системы WCF воздействует много различных факторов, таких как значения свойств `InstanceContextMode`, `ConcurrencyMode` и режим регулирования. Но при настройке вы всегда должны сначала придерживаться значений по умолчанию или рекомендуемых значений. Например, желательно использовать класс `DataContractSerializer`, а также режимы `PerCall` и `ConcurrencyMode.Single`. Изменять настройки по умолчанию имеет смысл только тогда, когда вы замечаете, что эти значения приводят к проблемам. При настройке всегда важно иметь под рукой надежный числовой материал, подтверждающий, привели ли ваши изменения к улучшению. Как уже упоминалось, платформа WCF предоставляет набор счетчиков производительности для общего мониторинга.

Балансировка нагрузки

Типичный случай масштабирования подразумевает добавление (по мере необходимости) в вашу среду большего количества серверов, чтобы они взяли на себя часть эксплуатационной нагрузки других серверов. Но балансировка нагрузки возможна только тогда, когда клиентские вызовы могут быть переданы на обработку любому серверу. Как правило, это возможно без любых проблем в случае режима `PerCall` без ориентированного сеанса протокола. Сообщения SOAP передаются на любой сервер, где создается новый объект для этого вызова. Если нужно, необходимые данные загружаются из другого источника данных (например, сервера SQL Server). После того как работа завершена, объект на стороне сервера удаляется. В случае режимов `InstanceMode.PerSession` и `InstanceMode.Single` или ориентированного на соединение протокола балансировка нагрузки со стандартными технологиями WCF невозможна. В режиме `PerSession`, в частности, платформа WCF неспособна поменять переменные сеанса на центральное состояние или сервер баз данных, в отличие от классического состояния сеанса ASP.NET.

6

Служба рабочего процесса

В ЭТОЙ ГЛАВЕ...

- Введение в службу рабочих процессов
- Обеспечение корреляции сообщений
- Настройка службы рабочих процессов
- Внедрение службы рабочих процессов

Рабочие процессы разработки программного обеспечения представляют собой эффективный инструмент для устранения проблемы или достижения целей проекта. Разделив их на более мелкие и более управляемые фрагменты, можно обеспечить их координацию в рамках единого процесса.

Рабочие процессы обеспечивают повторное использование кода и его эксплуатацию — два основных критерия качества, принятых в индустрии программного обеспечения. Это достигается благодаря тому, что рабочие процессы состоят из небольших фрагментов, которые можно повторно использовать в нескольких процессах и легче эксплуатировать. Эти фрагменты, или повторно используемые компоненты, называются действиями и обычно описывают работу, выполняемую людьми или программами, участвующими в технологическом процессе.

При разработке рабочих процессов предприятия с помощью подхода “снизу-вверх” их сначала представляют в виде совокупности элементарных операций для управления потоком выполнения программы, обмена информацией с внешней средой и изменения внутреннего состояния самих рабочих процессов. Позднее эти рабочие процессы можно преобразовать в действия и использовать в качестве компонентов для создания других рабочих процессов, координирующих работу на более высоком уровне. В этом случае процесс проектирования можно выполнять рекурсивно на разных уровнях абстракции.

Эту же парадигму проектирования можно применить для разработки служб. Службу можно реализовать в виде сочетания элементарных действий, образующих рабочий процесс, или в виде рабочего процесса, организующего другие службы для достижения поставленной цели.

Например, служба для подачи новой заявки в систему управления снабжением может потребовать координации с другими службами для проверки или обновления уровня запасов заказанного товара. Всю работу, необходимую для координации этих служб, можно описать как простой рабочий процесс.

Однако реализация службы в виде рабочего процесса — не такая простая задача, как кажется. Для этого необходимо разрешить много проблем, и без применения правильной технологии достижение поставленной цели может оказаться невозможным.

- Продолжительность долговременного процесса, предусматривающего взаимодействие с людьми, может измеряться часами и даже сутками. В качестве примера можно назвать службу, реализующую рабочий процесс рассмотрения документов. Для таких служб необходим механизм, позволяющий сохранять состояние рабочего процесса между разными моментами вызова службы. Невозможно навсегда сохранить это состояние в памяти, поскольку она представляет собой непостоянное хранилище информации. Кроме того, при одновременном выполнении многих рабочих процессов может возникнуть проблема, связанная с падением производительности работы приложения.
- В службу может поступать и из нее может исходить много сообщений, отражающих происходящие события. Необходим механизм корреляции всех этих сообщений, позволяющий возобновлять выполнение рабочего процесса на правильно выбранном этапе и в правильно определенном состоянии.

Workflow Foundation (WF) — это технология, позволяющая разрешить указанные проблемы. К счастью, с появлением новых возможностей для создания служб, использующих рабочие процессы, эта технология была тесно интегрирована с платформой WCF в пакете .NET Framework 3.5. Платформа WCF обеспечивает необходимую инфраструктуру для обеспечения взаимодействия рабочих процессов с внешней средой посредством обмена сообщениями.

В данной главе будет показано, как эти отношения были улучшены в версии 4.0 благодаря появлению новых функциональных возможностей, таких как корреляция сообщений на основе бизнес-логики, декларативная служба, а также новая модель выполнения действий и библиотека действий.

Структуры службы рабочего процесса

Служба рабочего процесса (workflow service), созданная по технологии WF, имеет состояние, получает входные сообщения извне и посылает исходящие сообщения во внешнюю среду, а также выполняет код, описывающий работу службы. Она ничем не отличается от обычной службы, обладающей состоянием и сохраняющей коллективно используемое состояние между разными моментами выполнения операций на протяжении своего существования.

На рис. 6.1 показана структура простой службы рабочего процесса. Каждая служба рабочего процесса имеет некое внешнее действие, представляющее собой рабочий процесс, состоящий из всех остальных действий. Это внешнее действие может иметь разные формы и определять поток выполнения процесса с помощью разнообразных внутренних действий. Пакет WF 4.0 поставляется с двумя встроенными действиями для

моделирования рабочих процессов: последовательность, которая проходит от одного действия к следующему, пока не будет выполнено последнее действие, и блок-схема, которая напоминает концепцию, используемую многими аналитиками и проектировщиками при создании решений или описании бизнес-процессов.

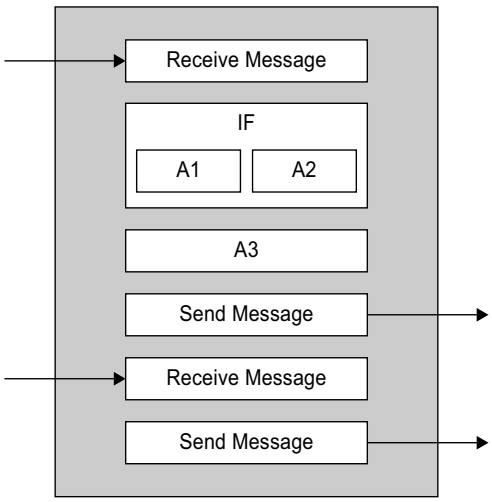


Рис. 6.1. Структура службы рабочего процесса

В этом примере выполнение рабочего процесса начинается с действия `Receive Message`, которое заключается в получении сообщения из внешней среды с помощью конечной точки WCF. За ней следует действие ветвления, которое проверяет условие перед выполнением одного из действий (A1 и A2 на рис. 6.1), определенного на каждой из ветвей. За действием ветвления следует выполнение следующего пользовательского действия (A3), а в заключение во внешнюю среду отправляется сообщение с помощью действия `Send Message` (которое может представлять собой вызов другой службы). Для получения следующего сообщения используется действие `Receive Message`, а за ним следует завершающее действие `Send Message`, оправляющее окончательный результат работы службы.

Как видим, каждое действие в этом примере представляет собой фрагмент типичной службы. Основное отличие заключается в том, что встроенные элементы языка, которые характерны для традиционных программ, заменены классами действий, предоставленными библиотекой действий платформы WF.

Следует понимать, что система выполнения платформы WF ничего не знает ни о внутренней реализации действий, которые она выполняет, ни об их поведении. Она может лишь выполнять одно действие за другим, пока не выполнит последнее действие. Кроме того, в некоторых точках она может сохранять состояние рабочего процесса.

Может возникнуть вопрос: зачем нужна служба рабочих процессов, если точно такую же службу можно реализовать вручную с помощью кода? Дело в том, что технология WF имеет несколько преимуществ.

- Наиболее очевидным преимуществом инфраструктуры WF является автоматическая поддержка долговременных рабочих процессов. Выполнение долговременных рабочих процессов может быть приостановлено или возобновлено во многих точках, поэтому необходимо сохранять внутреннее состояние и действия.

Платформа WF автоматически сохраняет состояния рабочего процесса с помощью конфигурируемых провайдеров. Следовательно, пользователь может не беспокоиться об этих деталях.

- Реализация сложной службы, организовывающей работу нескольких других служб с определенной степенью параллелизма, позволяет упростить использование рабочего процесса. Инфраструктура WF автоматически обрабатывает все аспекты, связанные с корреляцией сообщений. Та же реализация, реализованная исключительно с помощью кода, может потребовать много вспомогательных инструкций и определенного знания особенностей параллельного программирования. Это необходимо для обеспечения эффективности при координации всех вызовов служб и корреляции всех ответов, которые платформа WF выполняет автоматически.
- Служба может потребоваться для того, чтобы осуществить определенные компенсационные действия или остановить выполнение при возникновении неожиданной ситуации или исключения во время работы. Например, служба, работа которой зависит от других служб, может не завершить выполнение процесса, если одна из эти служб окажется недоступной, что может привести к несогласованности данных. Если эта служба реализована с помощью рабочего процесса, то он сам остановит свое выполнение и возобновит работу, когда требуемая служба станет доступной.
- Декларативная модель и визуальный конструктор, предоставляемые платформой WF, позволяют легко создавать простые службы, не заботясь о деталях реализации механизма WCF ABC (Address, Binding и Contract). Пользователь может сосредоточиться на декларативной модели, которая просо создает переменные, обрабатываемые в рабочем процессе, и действия, которые инициализируют эти переменные и изменяют их значения, посылая их во внешнюю среду.

Декларативные службы

Одна из основных проблем, связанных с реализацией служб рабочих процессов на платформе .NET Framework 3.5, заключается в том, что интеграция между этими технологиями оставляет желать лучшего. Обычно все артефакты платформы WCF определяются с помощью модели программирования и конфигурации WCF, а рабочий процесс определяется с помощью другой модели. В результате возникает множество артефактов, которые необходимо развертывать и организовывать отдельно.

Однако на платформе .NET Framework 4.0 все артефакты WCF, такие как контракты и операции, можно определить, используя язык *XAML* (eXtensible Application Markup Language), причем рабочий процесс определяется на этом же языке. В результате возникает единственный артефакт — служба, основанная на языке XAML.

Иначе говоря, пользователь, по существу, создает модель службы, определяя то, что она должна делать, а не пишет код, чтобы указать, как она должна это делать, — это традиционный способ работы по технологии .NET.

В листинге 6.1 приведено представление XAML службы рабочего процесса с простой операцией `GetData`, получающей входной аргумент в виде переменной `data` и возвращающей строку (`data.ToString()`).

Как видим, все действия представлены в виде объектов, имеющих свойства, с помощью языка XAML.

Листинг 6.1. Представление на языке XAML


```
<WorkflowService ConfigurationName="Service1" Name="Service1" ....>

  <p:Sequence DisplayName="Sequential Service">
    <p:Sequence.Variables>
      <p:Variable x:TypeArguments="CorrelationHandle" Name="handle" />
      <p:Variable x:TypeArguments="x:Int32" Name="data" />
    </p:Sequence.Variables>

    <Receive x:Name="__ReferenceID0" DisplayName="ReceiveRequest"
      OperationName="GetData" ServiceContractName="contract:IService">
      <Receive.CorrelationInitializers>
        <RequestReplyCorrelationInitializer CorrelationHandle="[handle]" />
      </Receive.CorrelationInitializers>
      <ReceiveMessageContent>
        <p:OutArgument x:TypeArguments="x:Int32"> [data] </p:OutArgument>
      </ReceiveMessageContent>
    </Receive>

    <SendReply Request="{x:Reference Name=__ReferenceID0}"
      DisplayName="SendResponse">
      <SendMessageContent>
        <p:InArgument x:TypeArguments="x:String"> [data.ToString()] </p:InArgument>
      </SendMessageContent>
    </SendReply>

  </p:Sequence>

</WorkflowService>
```

Кроме того, контракт и операции для этой службы автоматически выводятся из определения действия `Receive`.

Огромным преимуществом полностью декларативной службы является возможность хранить законченное определение в репозитарии службы и иметь полный контроль над экземплярами рабочих процессов из среды хостинга, например инфраструктуры Windows AppFabric.

Завершенные службы рабочего процесса на платформе WF 4.0 можно по-прежнему разрабатывать с помощью императивных языков технологии .NET, например C# или VB.NET.

Платформа WF 4.0 позволяет по-новому создавать завершенные определения рабочих процессов с помощью кода и без использования визуального конструктора и компонентной модели .NET. Поскольку модель этого вида требует, чтобы пользователь определил каждое действие и его свойства с помощью кода, при увеличении количества действий и усложнении рабочих процессов могут возникнуть проблемы, связанные с их эксплуатацией, — пользователь теряет визуальное определение, которое обеспечивает визуальный конструктор.

Если следовать этому подходу, то необходимо создать действия, представляющие модель рабочего процесса, например `Sequence` или `FlowChart`; действия по пересылке сообщений для взаимодействия с внешними рабочими процессами, например `Receive` или `Send`; а также другие действия для реализации самой службы. Контракт и операции службы по-прежнему выводятся из действия `Receive` — здесь нет формального определения контракта, как в технологии WCF.

```
    private static WorkflowServiceImplementation CreateWorkflow()
    {
        var result = new Sequence();
        var receivedInput = new Variable <string> ();
        result.Variables.Add(receivedInput);

        var handle = new Variable <CorrelationHandle> ();
        result.Variables.Add(handle);

        var receive = new Receive()
        {
          OperationName = "HelloWorld",
          ServiceContractName = "HelloWorldService",
          Content =
            ReceiveContent.Create(new OutArgument <string>
            (receivedInput)),
          CorrelatesWith = new InArgument <CorrelationHandle> (handle),
          CanCreateInstance = true
        };

        result.Activities.Add(receive);

        var write = new WriteLine()
        {
          Text = new InArgument <string> (env =>
          string.Format("Hello World!!! '{0}'.",
          receivedInput.Get(env)))
        };

        result.Activities.Add(write);
        var reply = new SendReply()
        {
           Request = receive,
           Content = SendContent.Create(new InArgument <string> (env =>
             string.Format("Hello World!!! '{0}'.",
             receivedInput.Get(env))))
        };
       result.Activities.Add(reply);
       var service = new WorkflowServiceImplementation
       {
           Name = "HelloWorldService",
           Body = result
       };
    return service;
}
```

В этом примере мы получаем сообщение, как определено в свойстве `Receive Content`. Оно представляет собой строку, присвоенную переменной `receivedInput`. Содержимое переменной `receivedInput` выводится на печать и возвращается через свойство `Content` действия `SendReply`.

Действия SEND и RECEIVE

В технологии .NET Framework 3.5 появилось два новых действия: `Send` и `Receive`, позволяющие упростить интеграцию между службами WCF и WF. Используя эти действия, можно включать конечные точки службы WCF в свою службу WF, чтобы сделать ее доступной для внешней среды. С другой стороны, службу WF можно использовать для реализации технологии при создании служб WCF. Рабочий процесс, использующий эти действия, обычно известен как служба рабочего процесса, и именно этой терминологии мы будем придерживаться.

Служба WF на платформе .NET Framework 4.0 по-прежнему поставляется вместе с этими действиями для отправки и получения односторонних сообщений посредством службы WCF, но она также содержит два новых действия, `SendAndReceiveReply` и `ReceiveAndSendReply`, относящихся к более высокому уровню абстракции операций "запрос/ответ".

Обсудим эти четыре действия более подробно и покажем, как их можно использовать.

Действие Receive

Действие `Receive` определяет точку входа в рабочий процесс, который начинает получать сообщения из внешней среды. Эта точка входа имеет вид конечной точки службы WCF, связанной с конкретными контрактом и операцией. В технологии .NET Framework 4.0 это действие претерпело значительные изменения и было улучшено. В частности, были упрощены определения контракта и операции службы, а также другие связанные с этим аспекты, например корреляция сообщений.

Некоторые свойства этого действия описаны в табл. 6.1.

Таблица 6.1. Свойства действия Receive

Свойство	Описание
`DisplayName`	Информативное имя, идентифицирующее действие в рабочем процессе. Это имя используется для ссылки из других действий или связывает данное действие с действием `SendActivity`, отправляющим результат выполнения операции во внешнюю среду
`ServiceContractName`	Имя контракта службы WCF, связанной с данным действием. Имя, выбранное для этого контракта, имеет важное значение, поскольку оно представляет контракт службы, который будет использоваться ее клиентами
`OperationName`	Имя операции службы WCF, связанной с данным действием. Эта операция добавляется в контракт, указанный в свойстве. Имя операции имеет такое же значение, как и имя контракта, по тем же причинам — оно является одним из видимых элементов службы
`Content`	Параметры ввода или аргументы, которые получает операция. Это содержимое обычно отображается в переменную рабочего процесса, поэтому может использоваться другими действиями
`Action`	Действие SOAP, которое будет использоваться для доставки сообщений, предназначенных для операции службы WCF. Это значение обычно посылается клиентом в адресной строке `To`
`CanCreateInstance`	Это свойство указывает, может ли действие создавать новый экземпляр рабочего процесса при получении нового сообщения через соответствующую конечную точку службы WCF. Каждое определение службы рабочего процесса должно иметь одно

Окончание табл. 6.1

Свойство	Описание
	действие `Receive`, способное создавать экземпляры (`CanCreateInstance = true`), и это действие должно быть первым действием рабочего процесса
`KnownTypes`	Коллекция типов платформы .NET, представляющая существующие контракты данных или элементарные типы, которые необходимо связать с определением операции в качестве известных типов. Известный тип — это тип, который должен учитываться в ходе процесса сериализации/десериализации при отправке/получении сообщений
`ProtectionLevel`	Уровень защиты, применяющийся к определению операции. Как указывалось в главе о безопасности, единственными возможными значениями для этого свойства являются `None`, `Sign` и `SignAndEncrypt`
`SerializerOption`	Сериализатор сообщений, связанный с операцией. Возможные значения этого свойства: `DataContractSerializer` и `XmlSerializer`. Эти механизмы сериализации обсуждались ранее
`CorrelatesOn/CorrelatesWith` и `CorrelationInitializers`	Это свойства для разрешения и связывания обработчика корреляции с операцией. Корреляция сообщений будет обсуждаться в этой главе позднее

Отдельное действие `Receive` в службе рабочего процесса представляет собой одностороннюю операцию в контракте службы, если пользователь не определил новое действие `SendReply`. Это действие связывается с действием `Receive`, чтобы отправлять ответное сообщение вместе с результатами выполнения операции.

Действие `SendReply` невозможно просто перетащить из инструментальной панели на экземпляр рабочего процесса в визуальном конструкторе. Его можно только создать на основе существующего действия `Receive`, выбрав команду Create SendReply в контекстном меню действия `Receive`.

Действие `SendReply` (рис. 6.2) связано с действием `Receive`, имеющем имя `ApplyForJobReceive`, которое выполняет операцию `ApplyForJob` и присваивает содержимое сообщения входной переменной `JobApp`. Как показано на рисунке, действие `SendReplyToReceive` связано с действием `Receive` с помощью свойства `Request`, а содержимое ответного сообщения представляет собой обычную строку. В реальной реализации службы пользователь может свободно связывать со свойством `Content` переменную, значение которой вычисляется в ходе рабочего процесса.

Эквивалентный контракт службы WCF для действия `Receive`, изображенного на рис. 6.2, должен выглядеть следующим образом.

```
  [ServiceContract]
public interface IHRService
{
  [OperationContract]
  string ApplyForJob(JobApp application);
}
```

Свойство `Content` имеет также несколько параметров, на случай, если пользователь не желает присваивать ответное сообщение отдельной переменной. Их можно выбрать в окне `Content Definition` при работе с визуальным конструктором рабочего

процесса. На рис. 6.3 показана операция `Receive`, входная информация которой представлена в виде нескольких аргументов.

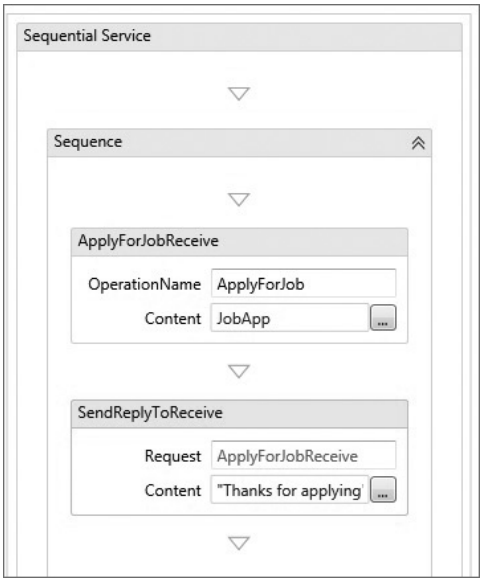


Рис. 6.2. Действие `SendReply`

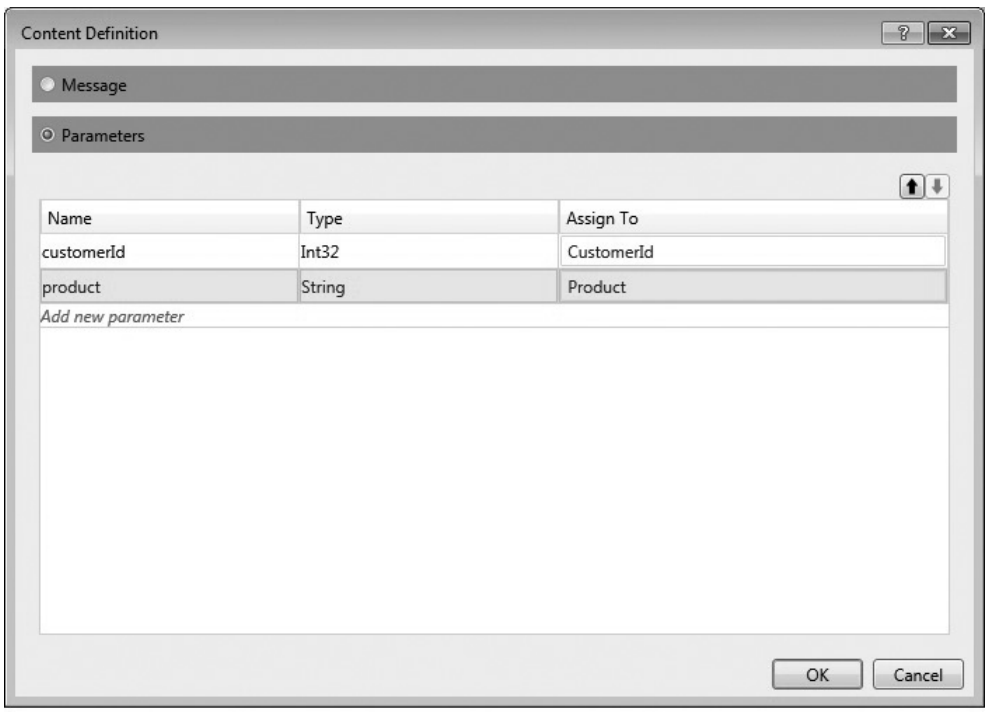


Рис. 6.3. Операция `Receive` *с несколькими аргументами*

Если пользователь не знаком с моделью рабочего процесса в рамках технологии WF 4.0, то для передачи значений в экземпляре рабочего процесса можно использовать переменные. Как и в любом языке программирования, переменная имеет область видимости, которая определяется точками, откуда переменную можно прочитать и/или откуда в переменную можно записать какую-либо информацию на протяжении времени ее существования, т.е. времени, пока она существует в памяти. Следовательно, совокупность всех переменных, находящихся в области видимости в конкретный момент времени, полностью определяет состояние рабочего процесса в данный момент. Это значительное улучшение механизма сохранения и загрузки рабочего процесса в память, обеспечивающее его персистентность. Механизм выполнения службы WF теперь может выполнять определенную оптимизацию и для этой цели использовать только переменные, находящиеся в области видимости. Прежде единственным способом разделения состояния между действиями, относящимися к рабочему процессу, было определение глобальных переменных, зависящих от самого рабочего процесса.

Действие Send

Действие `Send` представляет собой противоположность действия `Receive`. Действие `Receive` определяет точку входа службы WCF для получения сообщений, а действие `Send` позволяет рабочим процессам отправлять сообщения во внешнюю среду, чтобы выполнять внешние службы WCF, а также отправлять ответные сообщения вместе с результатами выполнения операции, адресуя их клиенту службы, связанному с действием `ReceiveReply`.

Аналогично тому, как отдельная операция `Receive` по умолчанию представляет собой одностороннюю операцию в контракте службы, действие `Send` не ожидает ответного сообщения, если она не связана с действием `ReceiveReply`.

Действие `ReceiveReply` является еще одним действием, на которое невозможно непосредственно сослаться в активном инструментальном окне и которое должно быть создано на основе существующего действия `Send`. Для этого следует выбрать команду Create ReceiveReply в контекстном меню действия `Send`.

Действие `ReceiveReply` (рис. 6.4) связано с действием `Send`, которое выполняет операцию `SubmitOrder`, а затем получает содержимое сообщения из переменной `Order`. Как показано на рисунке, действие `ReceiveReplyForSend` связано с действием `Send` с помощью свойства `Request`, а содержимое ответного сообщения возвращается в переменной `Result`.

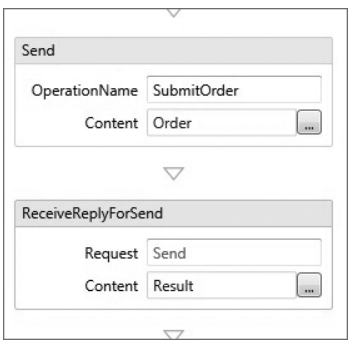


Рис. 6.4. Операция `ReceiveReply`

Некоторые свойства действия Send приведены табл. 6.2.

Таблица 6.2. Свойства действия Send

Свойство	Описание
Endpoint	Определение конечной точки, которая будет использоваться для обращения к службе. Это свойство состоит из двух частей: адреса конечной точки и соединения WCF. Адрес конечной точки — это идентификатор URI, представляющий собой адрес, где прослушивается служба. Соединение WCF — это одно из встроенных соединений, используемых для установки канала связи
EndpointConfigurationName	Информативное имя конечной точки в конфигурации службы WCF. Это свойство обеспечивает гибкость определения всех настроек конечной точки в файле конфигурации, например адреса, привязки и режима. Можно установить значение Endpoint или EndpointConfigurationName, но не оба одновременно
EndpointAddress	Адрес конечной точки службы, замещающий адрес, ранее определенный в свойстве Endpoint или в конфигурации службы WCF, если используется свойство EndpointConfigurationName
DisplayName	Информативное имя, идентифицирующее действие в рабочем процессе. Это имя используется также как ссылка на данное действие со стороны других действий, а также для связывания этого действия с действием ReceiveReply для получения ответа от службы
ServiceContractName	Имя контракта WCF, связанное с данным действием. Имя, выбранное для контракта, имеет важное значение, потому что оно представляет контракт службы, которое будет использовать служба WF. Это имя контракта состоит из двух частей: пространства имен и собственно имени контракта (например, http://wcfbook/}myServiceContract)
OperationName	Имя операции службы WCF, связанной с данным действием. Имя этой операции имеет важное значение по тем же причинам, что и имя контракта, — оно используется для обращения к службе
Content	Переменная рабочего процесса, которая отображается в запрос на выполнение операции службы
Action	Действие SOAP, включенное в адресную строку To запроса на выполнение операции службы
KnownTypes	Коллекция типов платформы .NET, представляющая существующие контракты данных или элементарные типы, которые необходимо связать с определением операции в качестве известных типов. Известный тип — это тип, который должен учитываться в ходе процесса сериализации/десериализации при отправке/получении сообщений
ProtectionLevel	Уровень защиты, применяющийся к определению операции. Как указывалось в главе о безопасности, единственными возможными значениями для этого свойства являются None, Sign и SignAndEncrypt
SerializerOption	Сериализатор сообщений, связанный с операцией. Возможные значения этого свойства: DataContractSerializer и XmlSerializer. Эти механизмы сериализации обсуждались ранее
CorrelatesWith и CorrelationInitializers	Свойство для разрешения и связывания обработчика корреляции с операцией. Корреляция сообщений будет обсуждаться в этой главе позднее

Для инициализации переменной, связанной со свойством `Content` действия `Send`, полезно применять действие `Assign`.

Действие `Assign` — это новое действие, появившееся в версии WF 4.0 для присваивания или инициализации значения переменной рабочего процесса. Оно состоит из двух частей: определения переменной, которой присваивается свойство `To`, и выражения для инициализации переменной в свойстве `Value`.

Выражения для инициализации переменной могут быть простыми, например скалярным значением, или более сложными, включающими функции или другие переменные, уже определенные в рабочем процессе. Сложные выражения должны определяться с помощью языка VB, поэтому, чтобы их правильно определить, необходимо знать этот язык.

На рис. 6.4 показан пример, в котором действие `Send` использует переменную `Order` для выполнения операции `SubmitOrder`. Поскольку эта переменная должна быть инициализирована до выполнения операции, можно использовать действие `Assign`.

Присваивание значения переменной `Order` можно выполнить с помощью следующего выражения.

```
Order = New Order()
With {.ProductName = "foo", .Quantity = 10}
```

Легко заметить, что для создания нового объекта класса `Order` здесь использована синтаксическая конструкция языка VB.

Поддерживает ли язык C# создание выражений? Нет. Основная причина — синхронизация. В окончательной версии технологии WF 4.0 разработчики языка VB предусмотрели программу синтаксического анализа для создания выражений, загруженную в оперативную память и выполняемую синхронно с WF 4.0.

Одно из ограничений текущей версии заключается в недостаточной поддержке для задания мандатов безопасности клиентов с помощью действия `Send`, в ситуациях, когда целевая служба должна провести аутентификацию вызывающего приложения/пользователя. В большинстве сценариев это можно сделать с помощью конфигурации WCF, используя привязку и описание режима. Таким способом можно задать лексемы мандатов `Windows`, `Certificates` и `Issued`, но, к сожалению, не лексему для имени пользователя. В этом случае пользователь не сможет воспользоваться внешней службой, пока не создаст пользовательское расширение или режим, задающее эти мандаты во время выполнения приложения.

Простой способ использования внешних служб

Другим интересным свойством технологии WF 4.0, значительно упрощающим разработку рабочих служб, является автоматическая генерация типизированных действий, использующих ссылки на внешние службы в проекте. Добавляя в проект ссылку на внешнюю службу, пользователь добавляет в инструментальное окно системы Visual Studio новые действия, связанные с использованием этой службы (по одной на операцию). Пользователь может просто перетащить одно из этих действий в рабочий процесс и заполнить все требуемые свойства, например, с помощью аргументов ввода и вывода, чтобы использовать службу, не заботясь о других деталях, которые ранее рассматривались при обсуждении действия `Send`. Преимущества автоматической генерации действий очевидны.

- Используются типизированные свойства для ввода и параметры вывода, которые операция службы связывает с действием.
- Все детали контрактов службы автоматически выводятся и включаются в генерируемые действия.

Пользователям рекомендуется всегда рассматривать альтернативы простейшему способу использования внешних служб, если ему не требуется более подробный контроль над деталями контракта клиентского WCF или настройками конечной точки.

Для использования операции `Notify` генерируются в контракт WCF и представление XAML действия.

```
[ServiceContract]
public interface IProductNotification
{
   [OperationContract]
   bool Notify(string product, int quantity);
}
```

```
<psa:Notify NotifyResult="[NotifyResult]" product="[PurchaseOrder.Product]"
quantity="[PurchaseOrder.Quantity]" />
```

Действия `SendAndReceiveReply` и `ReceiveAndSendReply`

Действия `SendAndReceiveReply` и `ReceiveAndSendReply` предназначены для упрощения определения операций "запрос/ответ".

Они являются эквивалентами комбинаций действий `Send/ReceiveReply` и `Receive/SendReply` соответственно, объединенных под оболочкой действия `Sequence`. Помимо того, что это последовательное действие является структурной единицей, оно определяет область видимости переменной, которую можно использовать для создания новых переменных.

Реализация первой службы рабочего процесса

Итак, мы получили первичные сведения о службах рабочих процессов и действиях, которые можно использовать в этих службах для получения сообщений или активизации внешних служб. Настало время перейти к более конкретному примеру, чтобы посмотреть на службу рабочего процесса в действии.

Этот пример иллюстрирует реализацию службы для ввода в систему заказов на поставку. Для достижения практических целей эта служба будет также вызывать внешнюю службу для каждого полученного заказа.

Начнем с создания библиотеки класса на языке C#, чтобы дать определение контракта и реализации внешней службы, активизируемой в службе рабочего процесса. Эта реализация довольно проста: она получает название заказанного товара, количество, указанное в заказе, и возвращает булево значение, означающее успех или неуспех операции (листинг 6.2).

Листинг 6.2. Контракт и реализация службы оповещения


```
[ServiceContract]
public interface IProductNotification
{
```

```
  [OperationContract]
  bool Notify(string product, int quantity);
}

public class ProductNotification : IProductNotification
{
  public bool Notify(string product, int quantity)
  {
     return true;
   }
}
```

Теперь у нас есть реализация внешней службы, и мы можем сосредоточиться на реализации первой службы рабочего процесса.

В системе Visual Studio 2010 есть шаблон WCF Workflow Service Application, относящийся к категории Workflow. Его можно использовать для создания служб рабочих процессов с нуля. При выборе и запуске этого шаблона создается новый веб-проект, содержащий шаблонное определение службы рабочего процесса, включающее в себя пару действий `Receive/Send`. Начнем реализацию своей службы рабочего процесса с удаления этих действий.

Эта служба вначале получает заказ на поставку, как это определено в контракте данных `PurchaseOrder` (листинг 6.3), с помощью нового действия `Receive`, которое должно быть создано в рабочем процессе.

Это действие, по существу, представляет собой точку входа в службу.

Листинг 6.3. Контракт и данные `Purchase Order`


```
[DataContract]
public class PurchaseOrder
{
  [DataMember]
  public string Customer { get; set; }

  [DataMember]
  public string Product { get; set; }

  [DataMember]
  public int Quantity { get; set; }

  [DataMember]
  public decimal UnitPrice { get; set; }
}
```

Теперь пользователь должен задать набор свойств действия `Receive` со значениями, указанными в табл. 6.3.

Таблица 6.3. Свойства действия `SubmitPOReceive`

Свойство	Значение
DisplayName	SubmitPOReceive
OperationName	SubmitPO
ServiceContractName	{http://wcfbook/}IPurchaseOrder
CanCreateInstance	True
Content (Message Data)	PurchaseOrder

Остальные свойства оставим равными значениям, установленным по умолчанию. Визуальный конструктор сообщит, что переменная `PurchaseOrder` не определена. Именно это нам необходимо сделать на следующем этапе.

Определять переменные в рабочем процессе можно на закладке Variables, которая обычно расположена у нижней границы рабочей области визуального конструктора. Определение переменной состоит из трех частей: имя, тип и область видимости. Как и в любом языке программирования, переменные имеют имя, а типом является любой допустимый тип CLR, входящий в состав платформы или созданный пользователем. В данном случае мы определяем переменную `PurchaseOrder`, имеющую тип `PurchaseOrder` (контракт данных, определенный в другом проекте). Область видимости этой переменной ограничена текущей службой рабочего процесса (службой `Sequential`). Легко видеть, что для получения всей информации существует один параметр контракта данных — `Message Data`. Если необходимо, чтобы сигнатура операции содержала несколько аргументов, следует использовать параметры и присваивать их соответствующим переменным или применять выражения языка VB для создания сложных выражений.

Теперь рабочий процесс готов получать сообщения, содержащие заказ на поставку с помощью операции `SubmitPO`. Если пользователь не определил действие `SendReply`, то эта операция будет считаться односторонней. На следующем этапе мы активизируем службу `ProductNotification` для каждого заказа. Существует простой способ включить вызов службы в существующий рабочий процесс — добавить ссылку на службу `ProductNotification`, как показано на рис. 6.5.

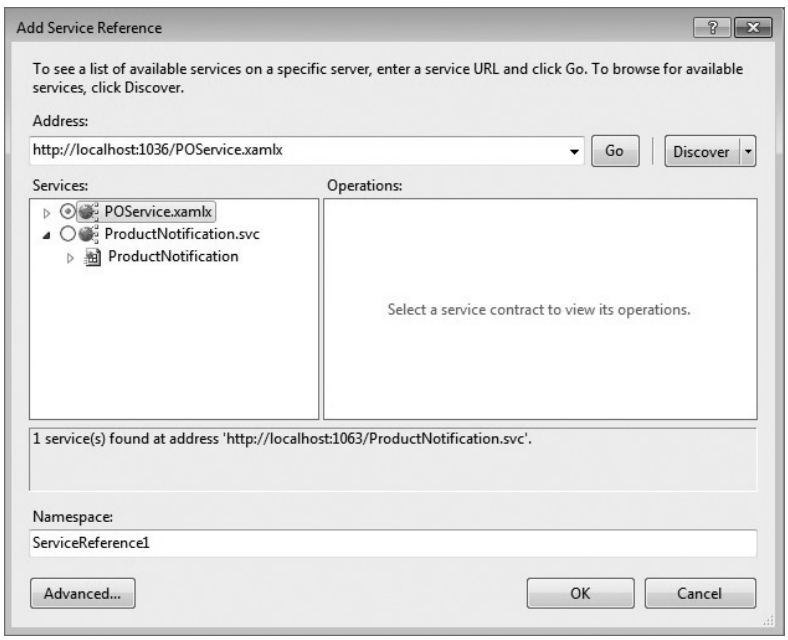


Рис. 6.5. Добавление ссылки на службу

Добавив ссылку в проект, пользователь увидит новое действие `Notify` с открытым интерфейсом, напоминающее операцию `Notify` в контракте `IProductNotify`. Она имеет два открытых свойства для задания названия товара и его количества соответственно. Третье свойство открывает доступ к значению, возвращаемому операцией.

Это действие можно перетащить на рабочую область после действия Receive и заполнить свойства значениями из табл. 6.4.

Таблица 6.4. Свойства действия Notify

Свойство	Значение
NotifyResult	NotifyResult
Product	PurchaseOrder.Product
Quantity	PurchaseOrder.Quantity

NotifyResult — это новая булева переменная, которая предназначена для получения результата операции. Аргументы ввода Product и Quantity инициализируются существующей переменной PurchaseOrder с помощью выражений.

В заключение необходимо создать новое действие SendReply на основе действия SubmitPOReceive, чтобы возвращать результаты обратно клиенту. Осталось только задать это действие в свойстве Content, чтобы использовать значения Message data и True. Итак, по существу, оно возвращает значение, служащее индикатором успешного выполнения операции.

В данный момент служба рабочего процесса готова к использованию, и любой клиент может получить доступ к службе WSDL, непосредственно просматривая файл POService.xamlx на соответствующем веб-узле (виртуальном каталоге IIS или веб-сервере Visual Studio). Это значит, что с точки зрения клиентского приложения данная служба функционирует так же, как любая другая служба WCF, а для генерации прокси-классов, использующих эту службу, можно применить утилиту svc (листинг 6.4.)

Листинг 6.4. Код, необходимый для использования службы рабочего процесса


```
PurchaseOrderClient client = new PurchaseOrderClient();
bool? result = client.SubmitPO(new ServiceReference.PurchaseOrder
{
  Customer = "John BarFoo",
  Product = "myproduct",
  Quantity = 10,
  UnitPrice = 1
});

Console.WriteLine(result.GetValueOrDefault());
```

Настройка службы рабочего процесса

Как у любой традиционной службы WCF, конфигурацию службы рабочего процесса можно отделить от реализации, определив ее либо в файле app.config для *локальных служб* (self-hosted services), либо в файле web.config для служб с размещением в системе IIS. Связь между реализацией службы рабочего процесса и соответствующей конфигурацией службы задается атрибутом ConfigurationName, который отображается в имя элемента конфигурации службы. Впрочем, конфигурацию службы рабочего процесса можно настроить непосредственно, как в обычной службе WCF, с помощью раздела конфигурации, описывающего модель службы. Следующий текст показывает, как атрибут ConfigurationName со значением "MyService" отображается в соответствующем разделе конфигурации службы.

```
<WorkflowService mc:Ignorable="sap" ConfigurationName="MyService"
Name="MyService"..>
</WorkflowService>

<system.serviceModel>
    <services>
      <service name="MyService" behaviorConfiguration="myService">
        <endpoint binding="basicHttpBinding"
                  bindingConfiguration="myService"
                  contract="IService"> </endpoint>
        </service>
      </services>
    <bindings>
        <basicHttpBinding>
           <binding name="myService">
              <security mode="None"> </security>
           </binding>
        </basicHttpBinding>
    </bindings>
    <behaviors>
        <serviceBehaviors>
          <behavior name="myService">
             <serviceMetadata httpGetEnabled="true"/>
          </behavior>
        </serviceBehaviors>
    </behaviors>
</system.serviceModel>
```

Имя контракта, необходимое для настройки конечной точки, должно совпадать с одним из имен контрактов, заданных для рабочего процесса `Receive`. В данном примере одно из действий `Receive` в рабочем процессе имеет значение `{[namespace]}IService` для свойства `ServiceContractName`.

При настройке службы рабочего процесса можно использовать новые усовершенствования конфигурации. По существу, шаблон системы Visual Studio для служб рабочего процесса автоматически включает файл конфигурации, который замещает лишь несколько настроек WCF, заданных по умолчанию, чтобы открыть доступ к конечной точке метаданных службы и отменить настройку `includeExceptionDetailsInFault`.

```
<system.serviceModel>
    <behaviors>
       <serviceBehaviors>
          <behavior>
            <!-- Чтобы избежать вскрытия информации метаданных, перед развертыванием
установите значение ниже в false и удалите конечную точку метаданных выше -->
            <serviceMetadata httpGetEnabled="true"/>
            <!-- Чтобы в целях отладки получить подробности исключения, в случае ошибки,
установите значение ниже в true. Установите его перед развертыванием в состояние false,
чтобы избежать вскрытия информации исключения -->
            <serviceDebug includeExceptionDetailInFaults="false"/>
          </behavior>
       </serviceBehaviors>
    </behaviors>
</system.serviceModel>
```

Настройки системы выполнения WF

Кроме соединений и настроек службы, необходимых для ее использования, в технологии WF 4.0 предусмотрен набор встроенных вариантов *поведения* (behaviors) для инициализации или замещения некоторых настроек системы выполнения WF. В этом разделе мы рассмотрим наиболее важные из них.

Поведение службы `WorkflowIdle`

Это поведение, как следует из его названия, дает пользователю контроль над моментами времени, когда неиспользуемые экземпляры рабочего процесса могут сохраняться или выгружаться из памяти. Оно содержит два свойства, `TimeToPersist` и `TimeToUnload`, которые устанавливают период времени для сохранения или выгрузки экземпляров из памяти. Следующий пример показывает, как задать конфигурацию системы выполнения, чтобы сохранять неиспользуемый экземпляр рабочего процесса каждые 10 секунд и выгружать их из памяти через минуту.

```
<workflowIdle timeToPersist="00:00:10" timeToUnload="00:01:00"/>
```

Поведение службы `SqlWorkflowInstanceStore`

Технология WF 4.0 имеет провайдера персистентности, `SqlWorkflowInstanceStore`, который выводится из абстрактного класса `System.Runtime.Persistence.InstanceStore` и реализует все абстрактные методы, позволяющие сохранить данные рабочего процесса либо на сервере SQL Server 2005, либо SQL Server 2008. Используя это поведение, можно изменять некоторые настройки, заданные для этого механизма по умолчанию, например, задать строку соединения или указать, следует ли удалить экземпляр из базы данных после его выполнения.

В следующем примере показана настройка строки соединения для данного провайдера персистентности.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="ServiceBehavior">
      <sqlWorkflowInstanceStore
        connectionString="Data Source=.\SQLExpress;Initial
        Catalog=WorkflowInstanceStore;Integrated Security=True">
      </sqlWorkflowInstanceStore>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Сценарии SQL для создания схемы базы данных и некоторые хранимые процедуры, необходимые для данного экземпляра провайдера, находятся в папке инсталляции платформы .NET (например, `C:\Windows\Microsoft.NET\Framework\v4.0.21006\SQL\en`). В данном случае нам нужны сценарии `SqlWorkflowInstanceStoreSchema.sql` и `SqlWorkflowInstanceStoreLogic.sql`.

Поведение службы `WorkflowUnhandledException`

Это поведение позволяет изменять действие на этапе выполнения при возникновении необработанного исключения в выполняемом экземпляре. По умолчанию система выполнения оставляет экземпляр в состоянии `Abandon`. Остальные значения для этого действия называются `Cancel`, `Terminate` и `AbandonAndSuspend`.

Когда управляющий узел службы вообще прекращает выполнение экземпляра рабочего процесса в памяти, возобновить работу впоследствии могут только экземпляры, имеющие состояние Abandon или AbandonAndSuspended. Кроме того, если рабочий поток был отменен, то вызываются все обработчики отмены, связанные с этим экземпляром, поэтому его работу можно прекратить очень аккуратно.

Поведение службы WorkflowRuntime

Поведение службы WorkflowRuntime позволяет изменять в ходе выполнения рабочего процесса некоторые специфичные настройки, которые используются для хостинга службы рабочего процесса. Это поведение также предоставляет экземплярам рабочего процесса возможность использовать дополнительные службы в ходе своего выполнения. Некоторые полезные атрибуты этого поведения приведены в табл. 6.5.

Таблица 6.5. Настройки поведения WorkflowRuntime

Настройка	Описание
CachedInstanceExpiration	Необязательная настройка, задающая максимальный период времени, в течение которого экземпляр рабочего процесса может оставаться в памяти в состоянии простоя, перед тем как он будет насильно выгружен, прекращен или остановлен
EnablePerformanceCounters	Не обязательная настройка, указывающая, включены ли счетчики выполнения. Эти счетчики сохраняют полезные статистические показатели о рабочем процессе, но могут тормозить производительность рабочего процесса в начале его выполнения и при запуске
ValidateOnCreate	Булева настройка, указывающая, выполняется ли проверка корректности определения рабочего процесса при открытии класса WorkflowServiceHost. По умолчанию она установлена, поэтому проверка корректности выполняется каждый раз, когда вызывается функция WorkflowServiceHost.Open. Если обнаруживается ошибка, передается исключение WorkflowValidationFailedException

Следующий фрагмент конфигурации иллюстрирует, как можно изменить некоторые настройки данного поведения.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="ServiceBehavior">
      <workflowRuntime name="WorkflowServiceHostRuntime"
                       validateOnCreate="true"
                       enablePerformanceCounters="true">
            <services>
            </services>
      </workflowRuntime>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Реализация корреляции сообщений

Как указывалось ранее, долговременные процессы являются типичными для служб рабочих процессов. Выполнение этих процессов обычно распадается на множество последовательностей, которые могут длиться несколько дней и даже недель. Эти

процессы могут вовлекать клиентов, которые соединяются со службой, выполняют работу, переводят рабочий процесс в новое состояние, а затем отсоединяются на неопределенное время, пока не возобновят соединение снова. Этот вид выполнения процессов порождает вопросы, которые необходимо решить, когда сообщение покидает экземпляр рабочего процесса, а затем в тот же процесс возвращается другое сообщение. Как сообщение может знать, в какой рабочий процесс оно должно вернуться? Что произойдет, если одновременно выполняются тысячи экземпляров этого процесса? Для этого рабочий процесс осуществляет корреляцию, т.е. пользователь может задать набор атрибутов для заданного сообщения, на основе которых система выполнения будет направлять сообщение в соответствующий экземпляр выполняемого рабочего процесса.

Атрибут корреляции в этом контексте может быть простым, как, например, уникальный идентификатор, а может быть сложным, как составной ключ, созданный на основе бизнес-концепций, например идентификатор потребителя или номер заказа на поставку.

Проблема корреляции в технологии WF 3.5 была решена с помощью клиентских приложений. После выполнения первой операции, созданной пользовательским рабочим процессом, к исходящему сообщению прикрепляется заголовок или пароль, содержащий идентификатор экземпляра рабочего процесса (его можно рассматривать как первичный ключ экземпляра рабочего процесса). Клиент должен сохранить этот идентификатор и присоединять его к сообщениям, чтобы следующие операции выполнялись в том же самом экземпляре рабочего процесса. Очевидно, что эта процедура является довольно сложной, поскольку клиент должен обладать определенной информацией о деталях реализации служб. Если клиент не знает, что он должен получить идентификатор из ответного сообщения и использовать его в следующем вызове, ничего работать не будет.

Это небольшая проблема, если и клиент, и служба проектируются одним и тем же разработчиком, но если клиент создан третьей стороной, ситуация усложняется.

В идеале сообщение, прикрепленное ко второй операции службы, должно иметь значение ключа, с которым можно вернуть экземпляр рабочего процесса обратно (атрибут корреляции). К счастью, разработчики технологии WF предусмотрели способ адресации сообщений экземплярам рабочих процессов. Этот способ называется *корреляцией на основе бизнес-логики* (content-based correlation).

В настоящем разделе рассматриваются два способа корреляции: старый, основанный на использовании уникального идентификатора, или корреляция, основанная на контексте, и новый, основанный на использовании ключа в самом сообщении.

Корреляция в действии

Механизмы корреляции основаны на использовании новых свойств, `CorrelatesOn`, `CorrelatesWith` и `CorrelationInitializers`, которые ранее обсуждались при изучении действий `Receive` и `Send`.

Механизмы корреляции определяют один или несколько обработчиков корреляции, представленных в рабочем процессе в виде обычных переменных типа `CorrelationHandler`. Они могут быть связаны с обменом сообщениями и инициализироваться в соответствии со стратегией, заданной функцией `CorrelationInitializer`. Система выполнения рабочих процессов использует эти обработчики корреляции, чтобы определить, какому экземпляру рабочего процесса следует направить входящие сообщения.

Инициализирующая функция корреляции определяет протокол, который система выполнения использует для инициализации обработчиков корреляции. Эта функция возвращает значение Context correlation initializer для маршрутизации, основанной на контексте; Request-Reply correlation initializer — для корреляции пары "запрос–ответ" в одном и том же канале; или Query correlation initializer — для маршрутизации, основанной на содержимом сообщений. Например, для использования механизма корреляции Query correlation initializer требуется также выражение, которое применяется к сообщениям и позволяет определить начальное значение для обработчика корреляции.

Два других свойства — CorrelatesOn и CorrelatesWith — используются только для маршрутизации, основанной на содержимом сообщений, и обсуждаются в следующих разделах.

Корреляция на основе контекста

Механизм контекстной корреляции использует протокол сообщений для передачи информации о маршруте между клиентом и службой рабочего процесса. Эта информация содержит как минимум уникальный идентификатор, известный также как ID экземпляра, который впервые генерируется, когда создается экземпляр рабочего процесса, и возвращается клиенту при первом сеансе обмена сообщениями. Именно клиентское приложение должно включать эту информацию во все последующие запросы, адресованные тому же самому экземпляру рабочего процесса. Существуют довольно сложные сценарии, в которых рабочий процесс может ожидать одно и то же сообщение при выполнении разных действий Receive одновременно. В этих сценариях одного идентификатора экземпляра недостаточно, и для направления сообщений правильно выбранному действию Receive необходим идентификатор сеанса. Идентификаторы instanceID и conversationID можно сделать частью контекстной информации.

Для передачи контекста между службой и клиентом реализация контекстного протокола обмена сообщениями в службе WCF может использовать пароль HttpCookie или заголовок SOAPHeader. Этот протокол изначально реализовывался на основе обычного канала, соединяющего контекст с уровнем приложения с помощью свойства ContextMessageProperty.

Значение этого свойства сериализируется либо как заголовок SOAP header, либо как пароль в заголовке Http до передачи сообщений по каналу связи. Технология WCF содержит также специальный соединительный элемент ContextBindingElement, цель которого — внедрить канал в уровень каналов WCF.

Этот соединительный элемент имеет открытое свойство ContextExchangeMechanism, принимающее только два значения — HttpCookie и SOAPHeader, — для выбора механизма сериализации.

Как правило, приложения не используют соединительные элементы непосредственно, поскольку они часто упакованы в элементы соединения более высокого уровня. Это также относится к соединительному элементу содержимого, поскольку технология WCF содержит набор специальных соединительных контекстных элементов, скрывающих от приложений детали механизма контекстного обмена сообщениями.

Список доступных контекстных элементов соединения приведен в табл. 6.6.

При рассмотрении табл. 6.6 может возникнуть вопрос: почему пропущен элемент NetMsmqBinding, обеспечивающий одностороннюю надежную и долговременную передачу сообщений службе рабочего процесса. Отметим, что для элемента ContextBindingElement необходим канал с интерфейсом IReplyChannel и элемент NetMsmqBinding на самом деле реализует интерфейс IInputChannel или

IOutputChannel. Но какой именно интерфейс он выберет для реализации, зависит от того, кем является пользователь — клиентом или службой. Если контекстный механизм обмена сообщениями работает в службе рабочего процесса, то это ограничение имеет смысл. Действие Receive, получающее экземпляр рабочего процесса, вызывается без контекста, и контекстный идентификатор экземпляра рабочего процесса возвращается как часть ответных сообщений. Следовательно, этот протокол отменяет все реализации односторонней передачи сообщений, а значит, и элемент NetMmsqBinding.

Таблица 6.6. Контекстные элементы соединения

Соединение	Описание
BasicHttpContextBinding	Вариант элемента BasicHttpBinding, поддерживающего протокол контекстного обмена сообщениями с помощью паролей на основе протокола http
WsHttpContextBinding	Вариант элемента WsHttpContextBinding, поддерживающего протокол контекстного обмена сообщениями с помощью заголовков SOAP либо паролей на основе протокола http
NetTcpContextBinding	Вариант элемента NetTcpContextBinding, поддерживающего протокол контекстного обмена сообщениями с помощью заголовков SOAP или паролей на основе протокола Tcp

Рассмотрим, как функционирует служба рабочего процесса с контекстной корреляцией сообщений на конкретном примере.

Этап 1. Реализация службы рабочего процесса

Служба рабочего процесса, использованная в этом примере, выполняет два действия Receive: первое действие заключается в том, что служба получает заказ на поставку, а второе — в том, что она подтверждает получение заказа.

Первое действие Receive, представляющее собой точку входа в службу, вначале получает заказ на поставку, как определено в контракте данных Purchase Order (листинг 6.5).

Листинг 6.5. Контракт данных PurchaseOrder


```
  [DataContract]
public class PurchaseOrder
{
  [DataMember]
  public string Customer { get; set; }

  [DataMember]
  public string Product { get; set; }

  [DataMember]
  public int Quantity { get; set; }

  [DataMember]
  public decimal UnitPrice { get; set; }
}
```

Установим свойства действия Receive равными соответствующим значениям, приведенным в табл. 6.7.

Таблица 6.7. Свойства действия `SubmitPOReceive`

Свойство	Значение
DisplayName	SubmitPOReceive
OperationName	SubmitPO
ServiceContractName	{http://wcfbook/}IService
CanCreateInstance	True
Content (Message Data)	PurchaseOrder

Оставим остальные свойства равными значениям, установленным по умолчанию, и определим новую переменную PurchaseOrder, предназначенную для получения содержимого запроса.

Наш рабочий процесс готов теперь получать сообщения, содержащие информацию о заказе на поставку, с помощью операции SubmitPO. Поскольку мы еще не определили действие SendReply, эта операция пока останется односторонней.

Следующее, что мы можем сделать, — использовать действие Assign, чтобы автоматически создать новый идентификатор заказа на поставку (сущность, которую можно хранить в состоянии рабочего процесса в виде переменной). Итак, мы можем определить новую переменную PONumber типа String и использовать следующее выражение в действии Assign для инициализации PONumber = 1. (Позднее эту подпрограмму можно будет заменить, чтобы новые номера присваивались автоматически.)

Зная номер заказа на поставку, можно создать действие SendReply на основе действия SubmitPOReceive, предназначенное для оправки переменной PONumber обратно клиенту.

Для этого нам необходимо установить свойство Content так, чтобы оно использовало значение содержания сообщения и значение PONumber и возвращало сгенерированный номер заказа на поставку.

Поскольку мы используем контекстную маршрутизацию, контекстный инициализатор действия Receive должен быть установлен как Context correlation initializer. Эту функцию можно связать с обработчиком корреляции handle1, который создается по умолчанию шаблоном системы Visual Studio.

Мы создали службу с единственной операцией, предназначенной для генерирования заказов на поставку. Теперь можно двигаться вперед к следующей операции, подтверждающей получение заказа на поставку. Для реализации этой операции необходима новая пара действий — Receive и SendReply.

Необходимо инициализировать новое действие Receive свойствами, приведенными в табл. 6.8.

Таблица 6.8. Свойства действия `ConfirmPOReceive`

Свойство	Значение
DisplayName	ConfirmPOReceive
OperationName	ConfirmPO
ServiceContractName	{http://wcfbook/}IService
CanCreateInstance	False

Это действие не получает никакой информации, поскольку нам не требуется ничего, кроме подтверждения заказа. У нас уже есть информация, являющаяся частью состояния рабочего процесса. Кроме того, стоит отметить, что свойство CanCreateInstance установлено равным false, чтобы повторно использовать существующий экземпляр рабочего процесса.

Действие SendReply для этой операции посылает ответ клиенту, сообщая информацию о результате подтверждения заказа. Для этой цели достаточно присвоить свойству Content следующее выражение: "The purchase order " + PONumber.ToString() + " was confirmed."

Этап 2. Настройка службы рабочего процесса

На следующем этапе необходимо настроить конфигурацию рабочего процесса, чтобы использовать одно из доступных контекстных соединений. Для простоты используем пример BasicHttpContextBinding. Как указывалось выше в разделе "Настройка службы рабочего процесса", имя службы рабочего процесса имеет значение для конфигурации.

Предположим, что мы присвоили нашей службе имя, и перейдем к листингу 6.6, в котором показана итоговая конфигурация с контекстным соединением.

Листинг 6.6. Конфигурация службы с контекстным соединением


```
<services>
  <service name="Service">
    <endpoint binding="basicHttpContextBinding"
      bindingConfiguration="myService"
      contract="IService"> </endpoint>
    </service>
  </services>
<bindings>
  <basicHttpContextBinding>
    <binding name="myService" contextManagementEnabled="true"> </binding>
  </basicHttpContextBinding>
</bindings>
```

Настройка contextManagementEnabled указывает, что наш рабочий процесс будет использовать контекстный протокол обмена, чтобы получать идентификаторы экземпляра и сеанса для корреляции сообщений.

Этап 3. Реализация клиентского приложения

Один из основных недостатков использования протокола контекстного обмена заключается в том, что клиенту необходимо знать внутренние детали реализации службы. Технология WCF предоставляет специальный интерфейс System.ServiceModel.Channels.IContextManager и соответствующую реализацию System.ServiceModel.Channels.ContextManager, которые автоматически обрабатывают большинство деталей на уровне канала. Этот класс можно использовать только тогда, когда заданы конфигурации контекстных соединений. Он представляет собой основную точку входа для задания контекстной информации — сценария с одной строкой, представляющей собой идентификатор экземпляра рабочего процесса.

Код, приведенный в листинге 6.7, описывает, как клиентское приложение может получить начальный идентификатор экземпляра и передать его последующим вызовам службы.

Листинг 6.7. Контракт данных для заказа на поставку


```
static void Main(string[] args)
{
  string instanceId;
```

```
  string poNumber = SubmitPO(out instanceId);

  Console.WriteLine("The PO Number is {0}", poNumber);

  string message = ConfirmPO(instanceId);

  Console.WriteLine(message);
}

private static string SubmitPO(out string instanceId)
{
  PurchaseOrder po = new PurchaseOrder
  {
    Customer = "Foo Bar",
    Product = "Product",
    Quantity = 10,
    UnitPrice = 5
  };

  ServiceClient client = new ServiceClient();

  string poNumber = client.SubmitPO(po);

  IContextManager contextManager =
    client.InnerChannel.GetProperty <IContextManager> ();
  IDictionary <string, string> context = contextManager.GetContext();

  instanceId = context["instanceId"];
  return poNumber;
}

private static string ConfirmPO(string instanceId)
{
  ServiceClient client = new ServiceClient();

  Dictionary <string, string> context = new Dictionary <string, string> ();

  context.Add("instanceId", instanceId);

  IContextManager contextManager =
    client.InnerChannel.GetProperty <IContextManager> ();

  contextManager.SetContext(context);

  return client.ConfirmPO();
}
```

В первом методе, `SubmitPO`, служба получает новый заказ, выполняя операцию с тем же именем, а идентификатор экземпляра извлекается из канала с помощью контекстного менеджера. Второй метод, `ConfirmPO`, получает идентификатор экземпляра как входной аргумент и вызывает операцию службы для подтверждения заказа путем передачи этого идентификатора через контекстный менеджер. Обратите внимание на то, что этот идентификатор заказа не передается всем службам, поскольку сам заказ

хранится как часть состояния рабочего процесса. Службе не обязательно выполнять дополнительный просмотр в операции `ConfirmPO`, чтобы извлечь заказ на поставку из хранилища, как это происходит в традиционных службах.

Этап 4. Настройка клиентского приложения

Конфигурация клиента должна содержать те же установки соединения, что и конфигурация службы. Настройки соединения, использованные в данном примере, приведены в листинге 6.8.

Листинг 6.8. Конфигурация клиента с контекстным соединением


```
<client>
  <endpoint address="http://localhost:1219/Service.xamlx"
    binding="basicHttpContextBinding"
    bindingConfiguration="myService" contract="ServiceReference.IService"
    name="BasicHttpContextBinding_IService" />
</client>
<bindings>
  <basicHttpContextBinding>
    <binding name="myService" contextManagementEnabled="true" />
  </basicHttpContextBinding>
</bindings>
```

Корреляция на основе бизнес-логики

Корреляция на основе бизнес-логики — новый механизм, появившийся в технологии WF 4.0 и предназначенный для передачи сообщений существующему экземпляру рабочего процесса на основе содержимого самих сообщений, например, номер заказа ли идентификатора клиента, а не на контекстной информации, которая используется в контекстных соединениях. Преимущества этого способа маршрутизации достаточно очевидны и упрощают разработку рабочих процессов и их настройку.

- Внутренние детали реализации службы не раскрываются перед внешним потребителем или клиентским приложением, потому что зависимость от контекстного протокола обмена полностью исключена.
- Поскольку контекстные соединения не нужны и информация о маршрутизации передается в самом сообщении, этот метод удобен для большего количества сценариев, чем при традиционной контекстной маршрутизации. Например, с помощью этого метода маршрутизации можно использовать сценарии, использующие надежные и долговременные механизмы передачи сообщений.

Этот механизм корреляции требует использования *функции инициализации Query*, связанной со всеми передачами сообщений. Она используется для инициализации обработчика корреляции выражением XPath, которое применяется к входящим и исходящим сообщениям.

После инициализации обработчика корреляции его можно связать с другим действием `Receive`, используя свойства `CorrelatesWith` и `CorrelatesOn`. Свойство `CorrelatesWith` получает переменную обработчика корреляции, а свойство `CorrelatesOn` позволяет спецификации нового выражения XPath получать одно или несколько значений из входящего сообщения, связанного с действием.

Система выполнения рабочего процесса узнает, как направить сообщения соответствующему экземпляру, сопоставляя значение из набора XPath в свойстве `CorrelatesOn` со значением, уже связанным с обработчиком корреляции (ранее инициализированным в другом действии `Receive` свойством `CorrelationInitializers`).

Рассмотрим предыдущий пример, но на этот раз с корреляцией, основанной не на контексте, а на бизнес-логике.

Этап 1. Реализация службы рабочего процесса

Как и в предыдущем примере, мы используем те же самые действия `Receive`, — первое действие, связанное с получением заказа на поставку, и второе, связанное с его подтверждением. Первое действие `Receive`, которое представляет собой точку входа в службу, вначале получает заказ на поставку, как определено в контракте данные `PurchaseOrder` (листинг 6.9).

Листинг 6.9. Контракт данных `PurchaseOrder`


```
 [DataContract]
public class PurchaseOrder
{
  [DataMember]
  public string Customer { get; set; }

  [DataMember]
  public string Product { get; set; }

  [DataMember]
  public int Quantity { get; set; }

  [DataMember]
  public decimal UnitPrice { get; set; }
}
```

Первое действие `Receive` должно быть инициализировано свойствами, указанными в табл. 6.9.

Таблица 6.9. Свойства действия `SubmitPOReceive`

Свойство	Значение
`DisplayName`	`SubmitPOReceive`
`OperationName`	`SubmitPO`
`ServiceContractName`	`{http://wcfbook/}IService`
`CanCreateInstance`	`True`
`Content (Message Data)`	`PurchaseOrder`

Остальные свойства оставим равными значениям, заданным по умолчанию, и определим новую переменную `PurchaseOrder`, предназначенную для получения содержимого запроса.

Затем можно использовать действие `Assign` для автоматического создания нового идентификатора запроса на поставку (сущности, представленной в состоянии рабочего процесса в виде переменной). Теперь мы можем определить новую переменную `PurchaseOrderId` типа `String` и использовать для ее инициализации в действии `Assign`

следующее выражение: `PurchaseOrderId = System.Guid.NewGuid().ToString()`. Это выражение инициализирует переменную `PurchaseOrderId` автоматически сгенерированным идентификатором `Guid`, что гарантирует ее уникальность.

Теперь наш рабочий процесс готов принимать сообщения, содержащее информацию о заказе на поставку посредством операции `SubmitPO`. Поскольку мы еще не определили действие `SendReply`, эта операция пока остается односторонней.

Получив номер заказа, можно создать действие `SendReply` на основе действия `SubmitPOReceive`, чтобы посылать содержимое переменной `PurchaseOrderId` обратно клиенту. Для того чтобы выполнить это, необходимо установить свойство `Content` так, чтобы оно использовало содержание сообщения и значение `PurchaseOrderId` и возвращало сгенерированный номер заказа.

Теперь необходимо определить свойство `CorrelationInitializers` для действия `SendReply`, чтобы использовать существующую переменную обработчика корреляции (для этой цели можно использовать переменную `__handle1`, созданную шаблоном системы Visual Studio), функцию `Query correlation initializer` и переменную `PurchaseOrderId`, которая становится содержимым действия. (Конструктор автоматически установит выражение XPath примерно так: `sm:body()/xgSc:SubmitPOResponse/xgSc:PONumber`.) Идентификатор заказа на поставку используется для корреляции последующих сообщений, которые клиент посылает службе рабочего процесса.

Мы создали службу с единственной операцией, предназначенной для генерирования заказов на поставку. Теперь можно двигаться вперед к следующей операции, подтверждающей получение заказа на поставку. Для реализации этой операции необходима новая пара действий — `Receive` и `SendReply`.

Необходимо инициализировать новое действие `Receive` свойствами, приведенными в табл. 6.8.

Таблица 6.10. Свойства действия `ConfirmPOReceive`

Свойство	Значение
`DisplayName`	`ConfirmPOReceive`
`OperationName`	`ConfirmPO`
`ServiceContractName`	`{http://wcfbook/}IService`
`CanCreateInstance`	`False`
`Content(Parameters)`	Новый параметр `PurchaseOrderId` типа `String`, который необходимо поставить в соответствие существующей переменной с тем же именем

Поскольку это действие не может создавать экземпляры, необходимо определить свойства `CorrelatesWith` и `CorrelatesOn` так, чтобы система выполнения рабочего процесса могла использовать их для направления правильному экземпляру рабочего процесса.

Свойство `CorrelatesWith` должно быть инициализировано переменной обработчика корреляции, ранее указанной в действии `SubmitPO` (например, `__handle1`). Другое свойство, `CorrelatesOn`, можно инициализировать параметром `PurchaseOrderId` (и опять же конструктор выведет выражение XPath автоматически).

Действие `SendReply` для этой операции посылает ответ клиенту, сообщая информацию о результате подтверждения заказа. Для этой цели достаточно присвоить свойству `Content` следующее выражение: `"The purchase order " + PONumber.ToString() + " was confirmed."`

Этап 2. Настройка службы рабочего процесса

Основное отличие от контекстной маршрутизации заключается в том, что маршрутизация на основе бизнес-логики не требует никакой конкретной конфигурации соединений, поэтому можно использовать конфигурацию WCF с настройками, заданными по умолчанию и автоматически сгенерированными шаблоном системы Visual Studio.

Этап 3. Реализация клиентского приложения

Преимущество маршрутизации, основанной на бизнес-логике, заключается в том, что все детали реализации службы совершенно скрыты от клиентского приложения, как это принято в традиционных службах в соответствии с четырьмя принципами организации служб. Следовательно, для использования этой службы пользователь может использовать традиционные прокси-классы WCF, и никакой дополнительный код для указания информации о маршрутизации не требуется.

Базовая реализация клиентского приложения, использующего описанную службу, представлена в листинге 6.10.

Листинг 6.10. Контракт данных `PurchaseOrder`


```
static void Main(string[] args)
{
  string purchaseOrderId = SubmitPO();

  Console.WriteLine("The PO Number {0} was submitted", purchaseOrderId);

  string message = ConfirmPO(purchaseOrderId);

  Console.WriteLine(message);
}

private static string SubmitPO()
{
  PurchaseOrder po = new PurchaseOrder
  {
    Customer = "Foo Bar",
    Product = "Product",
    Quantity = 10,
    UnitPrice = 5
  };

  ServiceClient client = new ServiceClient();

  SubmitPOResponse response = client.SubmitPO(po);

  return response.PurchaseOrderId;
}

private static string ConfirmPO(string purchaseOrderId)
{
  ServiceClient client = new ServiceClient();
  return client.ConfirmPO(new ConfirmPO { PurchaseOrderId = purchaseOrderId });
}
```

В первом методе, SubmitPO, служба получает новый заказ, выполняя операцию с тем же именем, а идентификатор экземпляра извлекается из ответа. Второй метод, ConfirmPO, получает идентификатор экземпляра как входной аргумент и вызывает операцию службы для подтверждения заказа путем передачи этого идентификатора в ответном сообщении. Как видим, все сложности, связанные с протоколом управления контекстом из клиентского приложения, полностью устранены, что очень хорошо.

Корреляция сообщения — не единственный вид корреляции, существующий в технологии WF. Как можно догадаться, рабочим процессам требуется поддержка корреляции при совместном использовании канала WCF между действиями Send и Receive. Несмотря на то что этот вид корреляции полностью осуществляется конструктором, пользователь должен знать о его существовании, чтобы использовать в коде рабочих процессов. Создавая действие SendReply на основе существующего действия Receive или делая то же самое с действиями ReceiveReply и Send, конструктор автоматически устанавливает связь между этими действиями с помощью функции Request-Reply Correlation Initializer.

Внедрение службы рабочего процесса

Внедрение службы рабочего процесса в технологии WF 4.0 также было улучшено. Кроме того, появилась возможность внедрять, поддерживать и эксплуатировать файлы с расширением .xamlx (это расширение используется для служб рабочего процесса и эквивалентно расширению .svc для обычных служб) в системе IIS и технологии Windows AppFabric.

Файл с расширением .xamlx, по существу, содержит полную реализацию и конфигурацию рабочего процесса в терминах языка XAML. Внедрение файлов с расширением .xamlx почти идентично внедрению файлов с расширением .svc, поскольку для хранения этих файлов необходим лишь виртуальный каталог, конфигурация службы в файле web.config и каталог bin со всеми сборками, от которых зависит служба.

Как и для любого приложения, использующего класс ServiceHost для внедрения обычных служб WCF в отдельный процесс за пределами системы IIS, существует эквивалентный класс, который делает с рабочим процессом то же самое, — класс WorkflowServiceHost. Он выводится из класса System.ServiceModel.ServiceHostBase и дополняет модель выполнения службы рабочего процесса всеми расширениями, от которых зависит рабочий процесс.

Класс WorkflowServiceHost содержит три перегруженных конструктора, обеспечивающих определение внедряемой службы рабочего процесса.

```
public WorkflowServiceHost(WorkflowService serviceDefinition,
params Uri[] baseAddresses);

public WorkflowServiceHost(Activity activity, params Uri[] baseAddresses);

public WorkflowServiceHost(object serviceObject, params Uri[] baseAddresses);
```

Первый конструктор получает экземпляр класса System.ServiceModel.Activities.WorkflowService, который представляет собой определение самого рабочего процесса. Используя этот новый класс, можно создавать, настраивать и получать доступ ко всем свойствам службы рабочего процесса. Два других конструктора создают и инициализируют новый экземпляр класса WorkflowService, используя либо

действие System.Activities.Activity (представляющее собой коренное действие службы), либо объект (который может быть или объектом класса WorkflowService, или экземпляром действия).

Определив рабочий процесс, можно настроить систему выполнения WCF так же, как и любую другую службу, используя код или конфигурацию. Это позволяет добавлять в систему выполнения конечные точки или функциональные возможности, а также модифицировать описание службы. Например, для того, чтобы добавить новые конечные точки службы, можно использовать либо коллекцию WorkflowService.Endpoints, как показано в следующем коде, либо непосредственно применить метод AddServiceEndpoint к экземпляру WorkflowServiceHost, как это принято для традиционных служб WCF.

```
var service = new WorkflowService();
service.Name = "HelloWorldService";
service.Body = CreateWorkflow();

var endpoint = new Endpoint
{
  AddressUri = new Uri("HelloWorld", UriKind.Relative),
  Binding = new BasicHttpBinding(),
  Name = "HelloWorldService",
  ServiceContractName = "HelloWorldService",
};

service.Endpoints.Add(endpoint);
```

Класс WorkflowServiceHost также содержит свойство DurableInstancingOptions для конфигурации провайдера персистентности, используемого системой выполнения рабочего процесса. Используя это свойство, можно заменить экземпляр InstanceStore, заданный по умолчанию, экземпляром, который хранит экземпляры рабочего процесса в базе данных сервера SQL (SqlWorkflowInstanceStore). Для этого достаточно написать несколько строк кода, как показано ниже.

```
var host = new WorkflowServiceHost(workflow, baseAddress);
 var connStr =
@"Data Source=.;Initial Catalog=WorkflowInstanceStore;Integrated Security=True";
var instanceStore = new SqlWorkflowInstanceStore(connStr);
host.DurableInstancingOptions.InstanceStore = instanceStore;
```

7

Основы безопасности WCF

В ЭТОЙ ГЛАВЕ...

- Принципы безопасности служб
- Механизм безопасности WCF

Обеспечение безопасности — критически важная часть любой технологии программирования или платформы для реализации службо-ориентированных приложений.

Как будет показано в данной главе, инфраструктура, необходимая для поддержки безопасности на уровне сообщений и служб, была разработана в технологии WCF с нуля. В этом смысле технология WCF обеспечивает разностороннюю и расширяемую модель безопасности, позволяющую разработчикам настраивать самые разные функциональные возможности системы выполнения рабочих процессов. В технологии WCF стало возможным гибкое внедрение расширений в подсистемы безопасности для поддержки существующих схем и сценариев безопасности, используемых для защиты служб.

В главе описаны многие функциональные возможности, обеспечиваемые технологией WCF в этой области, а также общие сценарии развертывания, которые могут понадобиться при разработке службо-ориентированных приложений. Кроме того, мы разберем основные понятия, связанные с безопасностью веб-служб. Если читатели уже имеют предварительное представление об этих концепциях, то изучение функциональных возможностей технологии WCF в области обеспечения безопасности будет более простым.

Эволюция систем безопасности в веб-службах

Безопасность всегда была строгим требованием, предъявляемым к веб-службам предприятий. Несомненно, безопасность и возможность взаимодействия сетей являются двумя ключевыми факторами, обеспечивающими освоение и успех веб-служб при разработке распределенных приложений в рамках предприятия.

Когда в конце 1990-х годов появились первые веб-службы, протокол SOAP не обеспечивал защиту секретных сообщений от взлома и даже не имел возможности шифровать сообщения, чтобы обеспечить их конфиденциальность. Все детали механизма обеспечения безопасности были делегированы на уровень транспорта, т.е. в тот момент — в протокол HTTP. По этой причине многие из существующих возможностей по обеспечению безопасности в протоколах HTTP и HTTPS были усилены средствами аутентификации клиентов и защиты сообщений. Например, посылка сообщения SOAP с помощью протокола HTTPS должна была гарантировать его конфиденциальность.

Однако опора на транспортный уровень для обеспечения безопасности сообщений связана с предварительной реализацией веб-службы для транспортного уровня, что создало серьезные препятствия для независимости от платформы протокола SOAP. Предприятия стали предъявлять веб-службам новые требования, поэтому под эгидой так называемых WC–* спецификаций появилось новое поколение веб-служб.

Спецификации WS–*, которые называют также *спецификациями веб-служб* (web services specifications), представляют собой набор протоколов и спецификаций, созданных ведущими компаниями в области программной индустрии, такими как Microsoft, IBM, VeriSign, SUN и др. Эти спецификации осиливают существующие функциональные возможности веб-служб, используя расширения первоначальной спецификации протокола SOAP.

Возвращаясь к вопросу безопасности, отметим, что одной из наиболее важных спецификаций WS–*, относящихся к этой области, является WS-безопасность.

Эта спецификация была опубликована 19 апреля 2004 года и была включена в новую модель обеспечения безопасности веб-служб на уровне сообщений, расширяющую существующую спецификацию протокола SOAP возможностями аутентификации клиентов и защиты сообщений.

Некоторое время спустя после анонса первых WS–* спецификаций компания Microsoft выпустила первую версию “облегченной” технологии для улучшения веб-служб WSE, предназначенную для расширения стека веб-служб на платформе .NET пользовательскими реализациями протоколов WS–*. Эта технология постепенно развивалась. Вслед за первой появились еще несколько версий, в которых было реализовано еще больше необходимых функциональных возможностей. Разработка технологии WSE была прекращена в 2006 году, когда была анонсирована технология WCF. Последняя версия WSE 3.0 содержала реализации хорошо известных протоколов, таких как WS-Security, WS-Addressing, WS-SecureConversation, MTOM и WS-ReliableMessaging. Если сравнить технологии WSE и WCF, то становится ясно, что технология WSE дала компании Microsoft возможность внедриться в мир WS–* спецификаций, и весь опыт и уроки, полученные при создании технологии WSE, был использован при проектировании и разработке многих функциональных возможностей технологий WCF и Windows Identity Foundation (WIF).

В настоящее время, по мере развития бизнеса, такие аспекты, как аутентификация и авторизация, медленно перемещаются в сторону интегрированных средств управления

доступом. Такие механизмы интерпретируют идентификацию пользователя как возможную службу, в которой функции аутентификации и авторизации представляют собой веб-службы, доступные любому приложению, существующему в рамках предприятия. Интегрированные средства управления доступом являются большим шагом вперед от замкнутых средств идентификации пользователей к более децентрализованным схемам аутентификации и авторизации. Компания Microsoft значительно продвинулась в этом направлении и недавно объявила о разработке платформы Geneva, представляющей собой часть стратегии, направленной на реализацию интегрированных средств управления доступом на платформе .NET.

Основные принципы безопасности веб-служб

В публикациях, посвященных безопасности веб-служб, обычно излагаются согласованные фундаментальные принципы, которые можно применить к любой распределенной системе передачи сообщений. Некоторые из этих принципов используются в этой главе, поэтому целесообразно повторить их, перед тем как погрузиться в детали технологии WCF.

Аутентификация

Аутентификация (authentication) — это процесс однозначной идентификации стороны, действующей как источник сообщений, поступающих в приложения или службы. В рамках главы эту сторону мы будем называть отправителем сообщения или клиентом службы.

В процессе аутентификации обычно решают два вопроса: “Кто вы?” и “Как вы это можете доказать?”

Отвечая на первый вопрос, отправитель должен предоставить некоторые свидетельства, подтверждающие его личность. Эти свидетельства могут иметь вид *нематериальных мандатов* (intangible credentials), например, в виде пары “имя и пароль пользователя”, билета протокола Kerberos, или токена с криптографической информацией, или материального удостоверения, например сертификата стандарта X509. С другой стороны, служба должна иметь механизм, проверяющий легитимность и благонадежность свидетельства. Отправитель считается успешно прошедшим аутентификацию, только если проверка свидетельства привела к благоприятным результатам.

Аутентификация клиента не является единственным сценарием аутентификации. Существуют ситуации, в которых отправитель желает верифицировать подлинность службы. Этот вид аутентификации называется аутентификацией службы и является проверенным механизмом для защиты от фишинг-атак.

Фишинг (phishing) в этом контексте представляет собой такую угрозу безопасности, при которой атакующая сторона делает доступной поддельную службу, сигнатура которой не отличается от сигнатуры оригинальной службы, чтобы перехватывать секретные или приватные данные о пользователе.

Допустим, некий магазин, продающий товары через Интернет, имеет специальную службу для посредников, поставляющих товары непосредственно клиентам. Эта служба получает конфиденциальную информацию о пользователе, например, о номерах его кредитной карточки и банковского счета. С одной стороны, посторонний интегратор

должен аутентифицировать сервер, прежде чем доверить ему конфиденциальную информацию о пользователе. С другой стороны, разработчик службы должен аутентифицировать пользователя, чтобы связать информацию о продаже с его профилем. Как видим, в этом типичном сценарии требуется взаимная аутентификация.

Авторизация

Авторизация (authorization) — это процесс, определяющий, какие ресурсы и операции системы могут быть доступными для пользователя. Иначе говоря, он определяет действия, которые разрешено делать пользователю.

Решения, полученные в ходе авторизации, большей частью основаны на свидетельствах подлинности аутентифицированного пользователя, например, на предъявляемой идентификационной информации пользователя или каком-либо другом свидетельстве, предоставленном клиентским приложением.

Продолжая пример, описанный в разделе, посвященном аутентификации, представим себе, что веб-сайт может предоставлять различные разрешения пользователям, обращающимся к службе с помощью специального ключа или атрибута, предоставленного третьей стороной.

Целостность сообщения

Целостность сообщения (message integrity) гарантирует, что данные, содержащиеся в сообщении, защищены от предумышленной или случайной модификации. Иначе говоря, она гарантирует, что данные, полученные службой, не были повреждены или искажены в пути.

Целостность данных в ходе их передачи в целом основана на технологиях криптографии, таких как цифровые подписи, хеширование и коды аутентификации сообщений. Целостность данных чрезвычайно полезна для предотвращения атак типа "человек посередине", в ходе которых злоумышленник преднамеренно начинает анализировать пакеты, передаваемые по сети, или модифицировать их перед тем, как они достигнут службы.

Обеспечение конфиденциальности сообщений

Обеспечение конфиденциальности сообщений (message confidentiality) — это процесс, гарантирующий, что данные, содержащиеся в сообщениях, остаются приватными и конфиденциальными, и их не может прочитать неавторизованная сторона. Как и целостность сообщений, конфиденциальность основана на криптографических технологиях, таких как шифрование данных. Для шифрования данных можно использовать разные алгоритмы, но большинство из них основано на асимметричных ключах, как, например, алгоритм RSA.

Шифрование данных позволяет противостоять атакам "человек посередине" или перехватам данных.

Безопасность транспорта и сообщений

Как указывалось в разделе "Эволюция систем безопасности веб-служб", для обеспечения безопасного обмена информацией между клиентскими приложениями, а также безопасности веб-служб, транспорта и сообщений обычно используются две модели.

Обе модели безопасности следуют определенным принципам, но разными способами (см. предыдущий раздел). Безопасность в этом контексте играет очень важную

роль, причем это в основном касается аутентификации и обеспечения целостности и конфиденциальности сообщений службы, проходящих через сеть.

Понимание различия между этими моделями поможет сделать правильный выбор, руководствуясь многими требованиями, предъявляемыми к реализации правильной схемы безопасности для службы.

Безопасность транспорта

Безопасность транспорта (transport security) обеспечивается функциональными возможностями, с помощью которых разнообразные средства передачи сообщений защищают сквозное соединение между отправителем сообщения или клиентским приложением и службой. Поскольку в оригинальной спецификации протокола SOAP вообще нет никаких схем безопасности, эта проблема очень важна для защиты соединений.

В транспортной безопасности все доступные механизмы аутентификации связаны с реализацией транспорта. Это приводит к большим проблемам для создания новых видов мандатов или расширения существующих.

То же самое происходит при обеспечении защиты сообщений; каждый транспорт имеет ограниченное количество встроенных опций. Большинство опций для защиты сообщений основано на сочетании протоколов *TLS* (Transport Layer Security) и *SSL* (Security Socket Layer), которые используются для шифрования и подписи сообщений, посылаемых по сети. Общим ограничением протоколов SSL/TLS является то, что они обеспечивают только двухточечную безопасность между двумя конечными точками — клиентом и сервером. Если сообщение пересылается с помощью разных посредников, они должны пересылать сообщение с помощью нового соединения SSL. Кроме того, после того, как сообщение покинет транспортное средство, оно больше не защищено.

Большим преимуществом транспортной безопасности над безопасностью сообщений является то, что участники не обязаны понимать, как работает механизм WS-Security вообще, что иногда представляет собой большое препятствие для обеспечения протокольного взаимодействия разных платформ или стеков веб-служб. Этот барьер является следствием двух факторов:

- сложности спецификации WS-Security;
- различий между окончательными реализациями веб-служб, разработанных разными поставщиками.

Безопасность сообщений

При использовании безопасности сообщений все метаданные, касающиеся безопасности, например цифровые подписи, шифрованные элементы, мандаты пользователей и криптографические ключи, содержатся в самом сообщении SOAP. Эта модель безопасности основана на спецификации WS-Security, которая описывает в основном механизм подписи или шифрования частей конверта SOAP с помощью разных алгоритмов и даже разных ключей.

Как и реализация XML, спецификация WS-Security представляет собой очень гибкое решение для поддержки разных мандатов или создания новых. Мандаты и ключи в этой спецификации представлены в виде токенов безопасности, представляющих собой обобщенную конструкцию XML, инкапсулирующую специфическую информацию о них. Кроме того, разные токены безопасности выводятся из существующих технологий обеспечения безопасности, таких как протокол Kerberos, сертификаты стандарта X509 certificates, а также имен и паролей пользователей. Они доступны в виде сопутствующих спецификаций, которые называются *профилями токенов* (token profiles).

Основное отличие от механизма безопасности транспорта заключается в том, что сообщения могут направляться другим службам или промежуточным системам без нарушения безопасности, — защита действует постоянно, независимо от того, покинуло сообщение транспортное средство или нет (рис. 7.1).

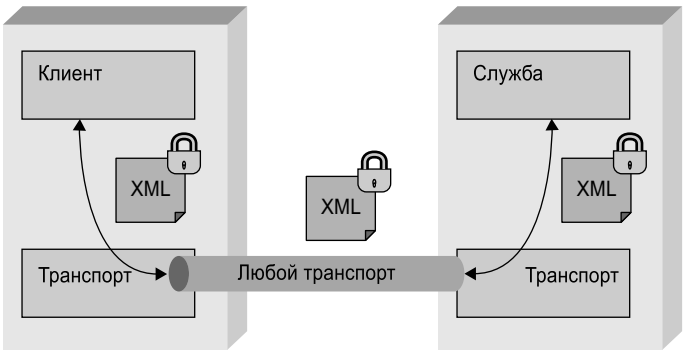


Рис. 7.1. Структура службы рабочего процесса

Обсудим, как работает механизм WS-Security в этом примере. Отправитель сообщения модифицирует его часть, заменяя тело зашифрованными данными, а остальные заголовки подписывает. Описание всех преобразований включается как часть специального заголовка протокола SOAP под названием “security”. Когда служба или отправитель сообщения получает сообщение, он может проанализировать этот специальный заголовок, чтобы выяснить, что тело было зашифровано, некоторые заголовки подписаны, а также какие криптографические ключи и алгоритмы использованы для выполнения этих операций. Если эти ключи принадлежат ему, он может расшифровать тело и проверить, что эти подписи корректны.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlsn:a="http://www.w3.org/2005/08/addressing"
xmlsn:u="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <s:Header>
    <a:Action s:mustUnderstand="1" u:Id="_2">
       http://tempuri.org/IHelloWorld/Hello
    </a:Action>
    <a:MessageID u:Id="_3">
       urn:uuid:4dabcfb1-6939-402d-919b-8e8486206e1f
    </a:MessageID>
    <o:Security s:mustUnderstand="1" xmlns:o="http://docs.oasis-open.org/
       wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
       <u:Timestamp u:Id="uuid-ea4659dc-85e5-4054-9b0c-c674f0128e7f-11">
          <u:Created> 2009-07-27T20:03:36.830Z </u:Created>
          <u:Expires> 2009-07-27T20:08:36.830Z </u:Expires>
        </u:Timestamp>
        <!-- Removed for simplicity -->
    </o:Security>
  </s:Header>
  <s:Body u:Id="_0">
    <e:EncryptedData Id="_1" Type="http://www.w3.org/2001/04/xmlenc#Content"
      xmlns:e="http://www.w3.org/2001/04/xmlenc#">
```

```
      <e:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc">
      </e:EncryptionMethod>
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <o:SecurityTokenReference xmlns:o="http://docs.oasis-open.org/wss/2004/01/
          oasis-200401-wss-wssecurity-secext-1.0.xsd">
          <o:Reference ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
            URI="#uuid-ea4659dc-85e5-4054-9b0c-c674f0128e7f-10"> </o:Reference>
        </o:SecurityTokenReference>
      </KeyInfo>
      <e:CipherData>
        <e:CipherValue> ..... </e:CipherValue>
      </e:CipherData>
    </e:EncryptedData>
  </s:Body>
</s:Envelope>
```

Предыдущий пример иллюстрирует, как механизм WS-Security применяется к сообщению SOAP. Как можно убедиться, тело было заменено зашифрованным разделом, содержащим ссылки на другие токены безопасности, содержащиеся в заголовках безопасности. Это ссылки, которые отправитель сообщения должен разрешить, чтобы получить доступ к криптографическим ключам.

Обзор механизма безопасности WCF

Итак, мы получили достаточно теоретической информации о механизме безопасности веб-служб и теперь готовы рассмотреть детали реализации служб рабочих процессов. В следующих разделах мы покажем, как использовать многие из доступных настроек безопасности в службе WCF, а также некоторые широко распространенные сценарии развертывания служб, в рамках которых принципы безопасности реализуются на практике.

Для защиты сообщений технология WCF поддерживает две традиционные модели безопасности: безопасность транспорта и безопасность сообщений, или гибридную модель, созданную на их основе (мандаты для безопасного транспорта и передачи сообщений).

Кроме того, существует множество готовых схем аутентификации и авторизации, удовлетворяющих требованиям традиционных сценариев. Если эти схемы не удовлетворяют требованиям пользователя, их можно легко настроить, поскольку расширяемость является одним из главных преимуществ технологии WCF в этой области.

Настройка механизма безопасности в технологии службы WCF

Привязка (binding) и поведения в службе WCF представляют собой точку входа в подсистему конфигурации и стратегии, влияющие на работу служб на этапе выполнения. Эта разнообразная и настраиваемая среда позволяет создавать стратегии безопасности и реализовывать их в службе на этапе выполнения. Выбор правильной схемы и стратегии обеспечения безопасности должен выражаться исключительно в терминах защиты сообщений, аутентификации и авторизации.

Кроме определения протокола взаимодействия и кодирования, привязки позволяют пользователю настраивать защиту сообщений и схему аутентификации. С другой стороны, поведения позволяют задавать другие виды настроек безопасности, например, мандаты клиентов и служб, средства проверки мандатов, стратегии авторизации и менеджеров.

Выбор соединения влияет также на стратегию обеспечения безопасности службы. Например, привязка `BasicHttp` поддерживает только традиционные протоколы веб-служб, следовательно, настраивать с помощью этой привязки можно только безопасность транспорта.

Все настройки привязок и поведений можно задать либо с помощью программирования в рамках объектной модели конфигурации, либо с помощью раздела конфигурации платформы .NET.

Для того чтобы избежать неверных решений, связанных с конфигурацией безопасности, разработчики технологии WCF ограничили количество настроек безопасности, которые можно использовать в разделе конфигурации. Мандаты имени пользователя и пароля представляют собой яркий пример этого явления. Было бы неудачным решением задавать какие-либо пароли в файле конфигурации, поэтому они доступны только в объектной модели конфигурации.

Чтобы еще больше упростить процесс конфигурации, все привязки поставляются с заранее заданными схемами конфигурации, удовлетворяющими большинство традиционных сценариев. В этом случае, если пользователь не задает свои собственные настройки, служба WCF пытается использовать схему безопасности, заданную по умолчанию.

Описание схемы безопасности для большинства привязок, заданной по умолчанию, приведено в табл. 7.1.

Таблица 7.1. Настройки безопасности, заданные по умолчанию

Привязка	Настройки
`WsHttpBinding`	Безопасность сообщения, обеспечиваемая протоколом аутентификации системы Windows (NTLM или Kerberos)
`BasicHttpBinding`	Нет настроек безопасности
`WsFederationHttpBinding`	Безопасность сообщения, обеспечиваемая интегрированным протоколом аутентификации (токены)
`NetTcpBinding`	Безопасность транспорта, обеспечиваемая протоколом Windows Authentication (NTLM или Kerberos)
`NetNamedPipeBinding`	Безопасность транспорта, обеспечиваемая протоколом Windows Authentication (NTLM или Kerberos)
`NetMsmqBinding`	Безопасность транспорта, обеспечиваемая протоколом Windows Authentication (NTLM или Kerberos)

Рассмотрим следующую конфигурацию, поддерживающую привязку `WsHttpBinding`.

```
<wsHttpBinding>
  <binding name="UsernameBinding">
     <security mode="Message">
        <message clientCredentialType="UserName"/>
     </security>
  </binding>
</wsHttpBinding>
```

В этом примере заданы настройки безопасности сообщений и профиль токенов имен пользователей. Остальные настройки безопасности для привязок заданы по умолчанию.

Теперь перейдем к изучению разных аспектов настройки механизма безопасности в службе WCF.

Режим безопасности

Настройки режима безопасности определяют два фундаментальных аспекта любой службы: модель безопасности для защиты сообщений и поддерживаемую схему аутентификации клиента.

Каждый режим безопасности имеет свой собственный механизм для передачи службе мандатов аутентификации, а значит, в зависимости от выбранной модели безопасности некоторые настройки аутентификации могут быть недоступными.

Например, интегрированная аутентификация поддерживается только механизмом аутентификации сообщений. Если необходимо обеспечить поддержку этой схемы безопасности, то нужно задать только настройку безопасности сообщений. Эта настройка доступна в модели конфигурации с помощью свойства `Mode` в элементе `security`.

Следующие примеры демонстрируют настройку режима безопасности для привязки `WsHttpBinding` с помощью раздела конфигурации WCF, а также эквивалентную версию с помощью кодирования.

```
<wsHttpBinding>
  <binding name="helloWorld">
    <security mode="TransportWithMessageCredential"> </security>
  </binding>
</wsHttpBinding>

WSHttpBinding binding = new WSHttpBinding();
binding.Security.Mode = SecurityMode.TransportWithMessageCredential;
```

Доступные для этого параметра возможности перечислены в табл. 7.2.

Таблица 7.2. Режимы безопасности

Режим	Описание
`None`	Служба доступна для всех, и сообщения в процессе их транспортирования передачи не защищены. При использовании этого режима служба уязвима для любых атак
`Transport`	Использует модель безопасности транспорта для аутентификации клиента и защиты сообщений. Этот режим имеет преимущества и недостатки механизма безопасности транспорта
`Message`	Использует модель безопасности сообщений для аутентификации клиента и защиты сообщений. Этот режим имеет преимущества и недостатки механизма безопасности сообщений
`Both`	Использует одновременно модели безопасности транспорта и сообщений для аутентификации клиента и защиты сообщений соответственно. Этот режим поддерживается только привязками MSMQ и требует одинаковых мандатов на обоих уровнях
`TransportWithMessageCredentials`	Защита сообщений обеспечивается транспортом, а мандат для аутентификации клиента службы является частью сообщения. Этот режим обеспечивает гибкость использования любых мандатов или токенов, используемых для аутентификации сообщений, в то время как аутентификация службы и защита сообщений осуществляется на уровне транспорта

Окончание табл. 7.2

Режим	Описание
`TransportCredentialOnly`	Использует механизм безопасности транспорта для аутентификации клиентов. Служба не аутентифицируется, а сообщения, включая мандаты клиентов, передаются в виде обычного текста. Этот режим безопасности может быть полезен в сценариях, в которых информация, которая передается между клиентом и службой, не является конфиденциальной, хотя мандаты выдаются всем

Уровень защиты

По умолчанию служба WCF шифрует и подписывает все сообщения, передаваемые по сети, чтобы обеспечить конфиденциальность и целостность данных. Иногда, если сообщение, передаваемое с помощью транспорта, не содержит конфиденциальной информации, пользователь может отключить шифрование и подпись сообщений, чтобы сохранить целостность данных, не беспокоясь об их секретности. Для этих сценариев служба WCF обеспечивает гибкость изменения уровня защиты, заданного по умолчанию, с помощью режима `Message Security`.

Уровень защиты можно настроить либо в определении контракта, либо для операций, что обеспечивает более плотный контроль. Если уровень защиты задан с помощью обоих механизмов, то уровень защиты, обеспечиваемый для операции, перекрывает существующее определение контракта.

Контракты сообщений также поддерживают перекрытие защитой, заданной для операции. Атрибут `[ProtectionLevel]` можно задать в самом контракте сообщения, или в специальном заголовке, или в теле сообщения.

Для настройки `ProtectionLevel` поддерживаются значения `None`, `Sign` и `EncryptAndSign`. Значение `None` отключает защиту сообщения. Значение `Sign` поддерживает только целостность сообщений, защищая их от любых возможных изменений. Значение `EncryptAndSign` обеспечивает конфиденциальность и целостность сообщений.

Уровень защиты может настраиваться только с помощью атрибутов в определении службы и не доступен в настройках конфигурации привязки. Следующие примеры иллюстрируют изменение настроек на разных уровнях (операции или контракта сообщения) в определении службы.

```
[MessageContract(ProtectionLevel = ProtectionLevel.None)]
public class HelloWorldRequestMessage
{
  [MessageHeader]
  public string SampleHeader { get; set; }

  [MessageBodyMember()]
  public string Message { get; set; }
}

[MessageContract]
public class HelloWorldResponseMessage
{
  public string ResponseMessage { get; set; }
}
[ServiceContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
```

```
public interface IHelloWorld
{
  [OperationContract(ProtectionLevel = ProtectionLevel.Sign)]
  HelloWorldResponseMessage Hello(HelloWorldRequestMessage request);
}
```

В результате сообщение запроса остается не защищенным, а ответное сообщение будет использовать уровень защиты, определенный только для операции подписи.

Набор алгоритмов

Настройка `algorithmSuite` является специфичной для безопасности сообщений и определяет набор алгоритмов, используемых для обеспечения целостности и конфиденциальности сообщений. Для практических целей необходимо понимать, что эта настройка, по существу, влияет на способ, которым служба WCF генерирует окончательные ключи для цифровой подписи и шифрования сообщений. Набор алгоритмов обычно представляет собой пару алгоритмов, состоящую из несимметричного алгоритма, для вывода симметричного ключа и симметричного алгоритма для шифрования и подписи сообщений.

Выполнение криптографических операций, таких как подпись или шифрование информации с помощью асимметричных ключей, связано с большими затратами компьютерных ресурсов, например центрального процессора или памяти. По этой причине служба WCF сначала генерирует и выводит симметричный ключ, используя асимметричный алгоритм, а затем использует симметричный ключ для эффективного выполнения криптографических операций.

Значение по умолчанию для этой настройки равно `Basic256`, что соответствует комбинации алгоритма `RSA-OAEP` в качестве асимметричного и алгоритма `AES256` в качестве симметричного.

Клиент или служба должны согласовать выбранный набор алгоритмов, чтобы обеспечить эффективное взаимодействие. Это очень важный аспект, поскольку некоторые платформы или стеки веб-служб поддерживают только конкретные наборы алгоритмов. Если возможность взаимодействия сетей не интересует пользователя, то эту настройку рекомендуется оставить заданной по умолчанию.

```
<netTcpBinding>
  <binding name="helloWorld">
    <security mode="Message">
      <message algorithmSuite="Basic256"/>
    </security>
  </binding>
</netTcpBinding>
```

Предыдущий сегмент конфигурации демонстрирует, как можно изменить настройку привязки `NetTcpBinding`, предназначенной для защиты сообщений.

Тип мандата клиента

Тип мандата клиента — очень важная настройка, определяющая схему аутентификации, используемую пользовательской службой. Она представляет собой тип мандатов, ожидаемых при аутентификации клиента или клиентского приложения.

Во-первых, следует знать, что опции этой настройки сильно зависят от выбранного режима безопасности и транспорта. Несмотря на то что опции безопасности сообщений, доступные в большинстве привязок, практически те же самые — `None`, `Windows`, `Username`, `Certificate` и `IssueToken`, — опции для безопасности транспорта ограничены поддержкой схем аутентификации на самом уровне транспорта.

Доступные опии для безопасности сообщений и безопасности транспорта приведены в табл. 7.3 и 7.4.

Таблица 7.3. Безопасность сообщений

Мандат	Описание типа
`None`	Служба не аутентифицирует клиентов. Это эквивалентно анонимной аутентификации
`Windows`	Служба аутентифицирует клиентов с помощью стандартного протокола аутентификации Windows: Kerberos или NTLM
`Username`	Служба аутентифицирует клиентов с помощью пары “имя–пароль”
`Certificate`	Служба аутентифицирует клиентов с помощью информации, представленной в сертификате протокола X509
`IssueToken`	Служба аутентифицирует клиентов с помощью токенов, выпущенных третьей стороной. Это характерно для интегрированных сценариев аутентификации

Таблица 7.4. Безопасность транспорта

Мандат	Описание типа
`None`	Эквивалентно опции `None` для настройки безопасности сообщений
`Windows`	Эквивалентно опции `Windows` для настройки безопасности сообщений
`Certificate`	Эквивалентно опции `Certificate` для настройки безопасности сообщений
`Basic`	Клиенты аутентифицируются протоколом `Http Basic Authentication`. Эта опция является специфичной для протокола `Http`
`Digest`	Клиенты аутентифицируются протоколом `Http Digest Authentication`. Эта опция является специфичной для протокола `Http`
`NTLM`	Клиенты аутентифицируются протоколом `Http Windows Integrated Authentication`. Эта опция является специфичной для протокола `Http transport`

Существует несколько исключений из предыдущей таблицы для привязок `NetMsqmBinding` и `NetNamedPipeBinding`.

Первая привязка заменяет настройку `clientCredentialType` эквивалентной настройкой `msmqAuthenticationMode`, поддерживающей режим безопасности транспорта `WindowsDomain` для аутентификации в системе Windows и режим проверки подлинности `Certificate` для аутентификации, основанной на сертификатах.

Привязка `NetNamedPipeBinding` обеспечивает только безопасность транспорта с аутентификацией Windows.

Эту настройку можно задать в модели конфигурации с помощью двух элементов: `Message` и `Transport`.

Служба WCF выбирает один из них на этапе выполнения в соответствии с заданным режимом безопасности.

```
<wsHttpBinding>
  <binding name="helloWorld">
```

```
    <security mode="Transport">
      <transport clientCredentialType="Windows"/>
      <message clientCredentialType="Windows"/>
    </security>
  </binding>
</wsHttpBinding>
```

В этом примере служба WCF использует определение типа мандата клиента в элементе транспорта, потому что установлен режим безопасности `Transport`. Элемент сообщения может по-прежнему существовать в конфигурации, но служба WCF его игнорирует.

Задание этой настройки с помощью кода осуществляется точно так же, как показано в разделе конфигурации.

```
WSHttpBinding binding = new WSHttpBinding();
binding.Security.Mode = SecurityMode.Transport;
binding.Security.Transport.ClientCredentialType = HttpClientCredentialType.Windows;
binding.Security.Message.ClientCredentialType = MessageCredentialType.Windows;
```

Аутентификация и согласование мандатов служб

Службы в технологии WCF должны предъявлять клиентам мандаты, обеспечивающие взаимную аутентификацию и защиту сообщений. Иначе говоря, прежде чем послать сообщение службе, клиент использует ее мандат для аутентификации, тем самым защищая такие аспекты сообщения, как конфиденциальность данных и их целостность.

В большинстве случаев мандаты автоматически согласовываются между клиентом и службой. Квитирование в рамках протокола SSL в механизме обеспечения безопасности транспорта представляет собой хороший пример такого согласования. Это квитирование позволяет серверу аутентифицировать себя перед клиентом путем предъявления открытого ключа X509, а затем позволяет клиенту и серверу совместно создать симметричный ключ, используемый для защиты сообщений в рамках последующего защищенного сеанса связи.

В некоторых схемах аутентификации, применяемых для защиты сообщений, служба WCF позволяет гибко заменять это поведение специфической настройкой `negotiateServiceCredentials`. По умолчанию служба WCF автоматически согласовывает мандаты на уровне сообщений, используя такие протоколы, как `TLSNego`, если установлен мандат клиента `None`, `Username` или `Certificate`, или протокол `SPNego`, если установлен мандат клиента `Windows`. Этот механизм согласования мандатов является довольно специфичным для службы WCF, поэтому пользователь должен отключить его, если необходимо обеспечить согласованное взаимодействие с другими стеками веб-служб или платформами. Это можно сделать следующим образом.

```
<wsHttpBinding>
  <binding name="helloWorld">
    <security mode="Message">
      <message clientCredentialType="Certificate"
               negotiateServiceCredential="false" />
    </security>
  </binding>
</wsHttpBinding>
```

Если эта настройка отключена, то при использовании сертификатных мандатов сертификат службы должен быть предъявлен всем клиентам еще до установления

соединения со службой, а клиенты должны сослаться на этот сертификат в своих конфигурациях.

```
<clientCredentials>
  <serviceCertificate>
    <defaultCertificate findValue="WCFService"
      storeLocation="LocalMachine" storeName="My"
      x509FindType="FindBySubjectName"/>
  </serviceCertificate>
</clientCredentials>
```

Аутентификация службы осуществляется платформой WCF в момент установки соединения путем сравнения мандатов с элементами конфигурации в конечной точке службы.

Если данный элемент в конфигурации не указан, то платформа WCF предположит, что используются значения по умолчанию. Например, если используются сертификатные мандаты, то платформа WCF проверит, соответствует ли имя субъекта в сертификате имени службы в Интернете.

В большинстве случаев конечную точку службы приходится задавать явно. Это можно сделать, добавив элемент идентификации в конфигурацию конечной точки.

```
<endpoint name="sampleProxy"
    address="http://localhost:8000/helloWorld/"
    bindingConfiguration="sampleBinding"
    behaviorConfiguration="sampleBehavior"
    binding="wsHttpBinding"
    contract="WCFBook.Samples.IHelloWorld">
    <identity>
      <dns value="WCFService"/>
    </identity>
</endpoint>
```

Эквивалентная версия в объектной модели выглядит следующим образом.

```
ServiceEndpoint ep = myServiceHost.AddServiceEndpoint(
typeof(WCFBook.Samples.IHelloWorld),
new WSHttpBinding(),
String.Empty);

EndpointAddress myEndpointAdd = new EndpointAddress(
new Uri("http://localhost:8000/helloWorld"),
EndpointIdentity.CreateDnsIdentity("WCFService"));

ep.Address = myEndpointAdd;
```

В предыдущем примере служба должна была предъявлять сертификат, содержащий имя субъекта, равное `WCFService`, чтобы платформа WCF правильно аутентифицировала ее.

Платформа WCF поддерживает пять типов сущностей для служб. Их использование зависит от сценария, который должен реализоваться, и требований безопасности, которые предъявляются службой. Все возможные типы сущностей перечислены в табл. 7.5.

Таблица 7.5. Типы сущностей

Сущность	Описание типа
`Domain Name System (DNS)`	Относится к сертификатам стандарта X509 или учетным записям Windows. Значение, заданное в этом элементе, должно соответствовать имени в учетной записи Windows или имени субъекта в сертификате. Если используется сертификат, то пока имя субъекта не изменится, проверка идентичности считается успешной
`Certificate`	Относится к сертификатам стандарта X509, зашифрованным в кодировке `Base64`. Если необходимо зашифровать весь сертификат, содержащий уникальную информацию, например индивидуальный код сертификата (`thumbprint`), этот тип сущностей является более точной альтернативой `DNS`. Обратной стороной этого типа является то, что пользователь должен "зашить" полное представление сертификата в конфигурацию платформы WCF
`Certificate Reference`	Почти совпадает с предыдущим вариантом. Основное отличие заключается в том, что этот тип сущностей позволяет указать имя сертификата и его место в хранилище сертификатов, а не кодировать представление мандата. Требует предварительного развертывания сертификата в хранилище сертификатов системы Windows
`RSA`	Задает сертификат ключа RSA. Этот вариант позволяет пользователю специально ограничивать аутентификацию отдельным сертификатом, основываясь на ключе сертификата. Как и при использовании опции `Certificate`, представление ключа должно быть представлено в кодировке `Base64`
`User Principal Name (UPN)`	Относится к аутентификации в системе Windows и задает основное имя пользователя (UPN), под которым выполняется служба. Эта опция используется по умолчанию, когда процесс службы не выполняется под именем одной из учетных записей системы. Иначе говоря, этот тип гарантирует, что процесс службы будет выполняться под именем определенной учетной записи Windows, в роли которой может выступать либо текущая регистрационная запись пользователя, либо учетная запись любого пользователя
`Service Principal Name (SPN)`	Относится к аутентификации в системе Windows и задает основное имя службы (SPN), связанное с учетной записью, под именем которой выполняется процесс службы. Эта опия используется по умолчанию, когда процесс службы выполняется под именем одной из системных учетных записей `LocalService`, `LocalSystem` или `NetworkService`. Платформа WCF автоматически согласовывает значение для этой сущности, если служба настроена на механизм аутентификации системы Windows, а свойство `negotiateServiceCredential` для механизма обеспечения безопасности сообщений задано равным `true`

Сеанс засекреченной связи

Сеансы засекреченной связи представляют собой последний аспект, который следует обсудить, прежде чем перейти к рассмотрению конкретных примеров в следующей главе.

Сеансы засекреченной связи (secure sessions), или *секретные разговоры* (secure conversations), позволяют ускорить время реакции службы, когда клиентское приложение требует обмена информацией со службой, в котором участвует несколько сообщений.

Если эта возможность отключена, то согласование мандатов и аутентификация осуществляются в ходе обмена первыми сообщениями между клиентом и службой. В противном случае эти два этапа выполняются при каждом вызове службы.

Не следует путать этот тип сеанса с типичными сеансами в рамках протокола `Http`. Сеансы засекреченной связи на платформе WCF инициируются клиентами и предназначены для поддержки общего контекста между клиентом и службой.

Эта функциональная возможность представляет собой специальную реализацию спецификаций `WS-SecureConversation` и `WS-Trust` в технологии WCF и, следовательно, требует использования механизма защиты сообщений.

В рамках сеанса засекреченной связи сначала клиент посылает службе специальное сообщение `RequestSecurityToken` (с действием SOAP `http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue`) с запросом на токен сеанса. Это сообщение является частью спецификации `WS-Trust` и содержит ссылку на мандаты клиента, которые будут использованы для создания токена сеанса.

```
<s:Envelope xmlns:s="..." xmlns:a="...">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue
    </a:Action>
    <a:MessageID> urn:uuid:d55bd2a7-bae8-4751-a010-07e95fd82ee2 </a:MessageID>
    <a:ReplyTo>
      <a:Address> http://www.w3.org/2005/08/addressing/anonymous </a:Address>
    </a:ReplyTo>
    <a:To s:mustUnderstand="1"> http://localhost/HelloWorld </a:To>
  </s:Header>
<s:Body>
  <t:RequestSecurityToken Context="uuid-f422503c-7974-42ab-9b8a-c330727290e9-1"
              xmlns:t="http://schemas.xmlsoap.org/ws/2005/02/trust">
    <t:TokenType> http://schemas.xmlsoap.org/ws/2005/02/sc/sct </t:TokenType>
    <t:RequestType>
      http://schemas.xmlsoap.org/ws/2005/02/trust/Issue
    </t:RequestType>
    <t:KeySize> 256 </t:KeySize>
    <t:BinaryExchange
      ValueType="http://schemas.xmlsoap.org/ws/2005/02/trust/spnego"
      EncodingType="http://docs.oasis-open.org/wss/2004/01/
        oasis-200401-wss-soap-message-security-1.0#Base64Binary">
    </t:BinaryExchange>
    </t:RequestSecurityToken>
  </s:Body>
</s:Envelope>
```

После того как служба получит сообщение и аутентифицирует мандаты клиента, она создает новый токен сеанса, который называется *контекстным токеном безопасности* (Secure Context Token – SCT), и ссылается на мандаты клиента и симметричный ключ для выполнения криптографических операций, например шифрования или подписи сообщений (тем самым обеспечивая конфиденциальность и целостность сообщений).

Служба в ответ посылает клиенту токен, используя сообщение `RequestSecurityTokenResponse` (с действием SOAP `http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Issue`), и следит за оригинальными мандатами, используя разные стратегии, такие как шифрованные идентификационные файлы протокола `Http` (cookies) или встроенный кэш.

```
<s:Envelope xmlns:s="..." xmlns:a="...">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Issue
    </a:action>
    <a:RelatesTo> urn:uuid:d55bd2a7-bae8-4751-a010-07e95fd82ee2 </a:RelatesTo>
  </s:Header>
  <s:Body>
  <t:RequestSecurityTokenResponse
    Context="uuid-f422503c-7974-42ab-9b8a-c330727290e9-1"
    xmlns:t="http://schemas.xmlsoap.org/ws/2005/02/trust"
    xmlns:u="http://docs.oasis-open.org/wss/2004/01/
      oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <t:BinaryExchange
      ValueType="http://schemas.xmlsoap.org/ws/2005/02/trust/spnego"
      EncodingType="http://docs.oasis-open.org/wss/2004/01/
        oasis-200401-wss-soap-message-security-1.0#Base64Binary">
    </t:BinaryExchange>
    </t:RequestSecurityTokenResponse>
  </s:Body>
</s:Envelope>
```

Таким образом, клиент позднее может защитить сообщение с помощью симметричного ключа, включенного в токен сеанса, или использовать токен сеанса в качестве мандата клиента.

Самые важные преимущества использования контекстного токена безопасности для защиты связи между клиентом и службой описаны ниже.

1. Время реакции службы увеличивается в три-четыре раза по сравнению с выполнением тех же действий с другими мандатами, поскольку клиент аутентифицируется только в первый раз, а для защиты линии связи используется симметричный ключ.
2. Токен действует короткое время, но его можно автоматически возобновлять. Вследствие этого клиентское приложение не обязано следить за оригинальными мандатами. Это важный аспект в ситуациях, когда оригинальные мандаты содержат конфиденциальную информацию, например имя пользователя или пароль.

С точки зрения клиентского приложения защищенный обмен информацией является полностью прозрачным, причем это свойство поддерживается платформой WCF на уровне канала. По этой причине токен сеанса используется повторно, пока клиентское приложение использует тот же самый экземпляр клиентского канала.

Если клиентский канал закрылся нормально, службе посылается сообщение, получив которое она закрывает сеанс и освобождает занятые ресурсы. Если канал был закрыт внезапно, то сеанс будет в конце концов закрыт службой после определенного периода простоя.

Это свойство устанавливается по умолчанию для всех привязок, поддерживающих безопасность сообщений (WS-Security). Пользовательская привязка также позволяет обеспечить сеанс засекреченной связи, установив значение `SecureConversation` для атрибута `authenticationMode`. Кроме того, пользователь может задать конфигурацию привязки, которая использовалась для согласования токена сеанса посредством элемента `secureConversationBootstrap`.

Следующие настройки демонстрируют, как установить сеанс засекреченной связи для привязки `wsHttpBinding`, а также эквивалентной версии пользовательской привязки.

```
<bindings>
  <wsHttpBinding>
    <binding name="ServiceBinding">
      <security mode="Message">
        <message clientCredentialType="Certificate"
           establishSecurityContext="true"/>
      </security>
    </binding>
  </wsHttpBinding>

  <customBinding>
    <binding name="ServiceBinding">

      <security authenticationMode="SecureConversation"
            requireSecurityContextCancellation ="false">
         <secureConversationBootstrap authenticationMode="MutualCertificate">
         </secureConversationBootstrap>
      </security>
      <httpTransport/>
    </binding>
  </customBinding>
</bindings>
```

Атрибут `requireSecurityContextCancellation` в предыдущей конфигурации указывает, должно ли клиентское приложение посылать сообщение о прекращении сеанса службе после закрытия клиентского канала. Если этот атрибут установлен равным `false`, то платформа WCF использует идентификационный файл (cookie) для отслеживания состояния токена сеанса, а не хранит его в памяти службы. Поскольку при каждой передаче сообщения, адресованного службе, ей пересылаются идентификационные файлы, они представляют собой идеальный инструмент для использования в веб, где применяются промежуточные серверы или балансировка нагрузки, и нет гарантии, что все сообщения будут обработаны на одном и том же сервере.

Клиентский канал невозможно использовать повторно для отправки нескольких сообщений одной и той же службе, поскольку при этом исчезают преимущества защищенного сеанса связи. Дополнительная нагрузка, связанная с первоначальным согласованием, может снизить общую производительность службы, поэтому следует рассмотреть возможность отключения этой функциональной возможности, когда это происходит.

8

Функционирование системы безопасности WCF

В ЭТОЙ ГЛАВЕ...

- Принципы аутентификации в технологии WCF
- Основы безопасности на основе утверждений
- Авторизация клиентских служб в технологии WCF

Предыдущая глава посвящена фундаментальным вопросам, влияющим на правильный выбор схемы безопасности в пользовательских службах WCF.

В центре внимания данной главы — применение этих концепций в реальных сценариях и примеры, которые будут выполнены шаг за шагом. Кроме того, в ней обсуждаются альтернативы, предлагаемые технологией WCF, для аутентификации и авторизации клиентов в ситуациях, когда в службу присылается мандат клиента, при этом основной упор делается на модель безопасности, основанную на утверждениях.

Введение в аутентификацию

Правильный выбор схемы аутентификации для пользовательской службы обычно основан на нескольких факторах, например: местоположение клиента в интранет или Интернет, возможность развертывать мандат у клиентов, а также количество потенциальных клиентов службы.

Как указывалось ранее, аутентификация в контексте технологии WCF обычно подразумевается взаимной.

Взаимная аутентификация (mutual authentication) — это двунаправленный процесс, в котором клиенты и служба аутентифицируют друг друга. Этот вид аутентификации чрезвычайно важен для служб, представленных в Интернете. Атакующая сторона

может оказаться способной обмануть одну из служб и перехватывать вызовы клиентов, чтобы получить доступ к конфиденциальной информации.

Мандат службы зависит от выбранных схем аутентификации клиента и режима безопасности. Как правило, если пользователь выбрал схему аутентификации клиента, например аутентификацию имени пользователя или сертификата в режиме обеспечения безопасности сообщений, то для взаимной аутентификации и защиты сообщений необходим сертификат службы. Если используется аутентификация системы Windows, то для аутентификации службы и защиты сообщений можно применять мандат системы Windows.

Поскольку количество возможных схем аутентификации, которые должна применять служба, определяется количеством всевозможных сочетаний типов мандатов клиентов и режимов безопасности, технология WCF предложила новый способ идентификации клиента с помощью унифицированной аутентификации на основе утверждений. В этом случае службы освобождаются от деталей реализации всех поддерживаемых механизмов аутентификации и полагаются только на утверждения, чтобы идентифицировать клиентов.

Модель идентификации на основе утверждений

Утверждение (claim) в рамках технологии WCF описывает часть идентификационной информации о субъекте (клиентской службе) в специфическом контексте, а также его индивидуальные права или действия, допустимые для данного ресурса.

Утверждение состоит из четырех частей.

- Тип утверждения.
- Информация или содержимое, специфичные для утверждения.
- Информация об издателе утверждения.
- Описывает ли утверждение сущность или возможности субъекта.

Информация об издателе также играет довольно важную роль в этой модели, поскольку служба может доверять или не доверять информации, представленной в утверждении, основываясь на данных о его издателе.

Рассмотрим пример из реального мира. Если вы показываете водительское удостоверение в винном магазине, чтобы доказать, что вы достигли совершеннолетия и имеете право покупать алкоголь, то вы предъявляете свидетельство о своей личности третьей стороне в виде утверждений. Каждое утверждение в лицензии сравнивается с конкретным фрагментом информации о вас или о самой лицензии, например, с именем, днем рождения или датой прекращения действия удостоверения.

В этом примере личность, продающую вам вино, вероятно, интересует лишь утверждение о вашем возрасте, которое можно вывести из информации о дате вашего рождения. Эта личность может также проверить, является ли ваше водительское удостоверение действующим и было ли оно выдано органом, заслуживающим доверия (аутентификация). Ситуация будет иной, если вы представите карточку, выданную органом, не заслуживающим доверия, т.е. поддельное удостоверение. Продавец в этом случае обязательно ее отвергнет.

В технологии WCF каждый тип успешно аутентифицированного мандата или токена безопасности превращается в набор утверждений и передается службе с помощью контекста безопасности, связанного с выполняемой операцией.

Эти утверждения не только содержат идентификационную информацию о клиенте, но и данные о том, что служба может принимать решения об аутентификации при выполнении операции.

В технологии WCF утверждение представляется в виде класса `System.IdentityModel.Claims.Claim`, который является частью сборки `System.IdentityModel`. Большинство классов, предназначенных для обработки разных аспектов аутентификации, в технологии WCF находятся именно в этой сборке.

```
[DataContract(Namespace = "http://schemas.xmlsoap.org/ws/2005/05/identity")]
public class Claim
{
  [DataMember(Name = "ClaimType")]

  public string ClaimType { get; set; }

  [DataMember(Name = "Resource")]
  public object Resource { get; set; }

  [DataMember(Name = "Right")]
  public string Right { get; set; }
}
```

Можно заметить, что этот класс снабжен атрибутами `DataContract/DataMember`, поэтому его можно передавать по сети, пересекая границы службы.

Свойство `ClaimType` является уникальной строкой идентификатора, которая, как правило, связана с предписанным идентификатором URI, представляющим этот вид информации в утверждении. Например, тип `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name` может представлять утверждение, содержащее имя пользователя.

Технология WCF изначально содержит набор предписанных типов утверждений в классе `System.IdentityModel.Claims.ClaimTypes`, но пользователь может без ограничений использовать любую схему для представления типов утверждений. Идея заключается в том, чтобы иметь возможность повторно использовать большинство из этих типов и создавать новые, когда необходимо представить типы, специфичные для бизнеса, к которому относится данная служба.

Некоторые из заранее определенных типов утверждений, которые можно найти в этом классе, приведены в табл. 8.1.

Таблица 8.1. Типы утверждений

Тип	Значение
Name	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
Surname	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname
E-mail	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress
Country	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/country

Свойство `Resource` ссылается на реальное значение утверждения. Поскольку это свойство представляет собой объект, с утверждением можно связать любой вид информации: от простого значения, например строки, до более сложных графических данных. Если необходимо сериализировать утверждения, то сначала следует убедиться, что значения, присвоенные им, также можно сериализировать.

Последнее, но не менее важное свойство, `Right`, указывает, представляет ли утверждение идентификационную информацию об аутентифицируемом объекте, например имя, или какую-то дополнительную информацию для принятия решения об авторизации, например адрес электронной почти пользователя, его номер телефона и т.д.

Эти два возможных значения свойства `Right` доступны в виде свойств в классе `System.IdentityModel.Rights` (табл. 8.2)

Таблица 8.2. Типы прав

Тип	Значение	Предназначение
`Identity`	`http://schemas.xmlsoap.org/ws/2005/05/identity/right/identity`	Утверждение о таких правах однозначно идентифицирует субъекта, которого оно описывает
`PossessProperty`	`http://schemas.xmlsoap.org/ws/2005/05/identity/right/possessproperty`	Утверждение о таких правах предоставляет дополнительную информацию о субъекте

Значения этого свойства представляют собой обычные строки, что характерно для типов утверждений, поэтому пользователь может предложить свою собственную схему описания утверждений о правах.

Технология WCF по умолчанию генерирует новый набор утверждений для каждого аутентифицированного токена или мандата клиента. Клиенты обычно посылают по одному токену безопасности на сообщение, который называется *основным* (primary) и может иметь любой тип мандата, доступный в технологии WCF или заданный пользователем. Однако клиент не ограничен условием посылать только один токен: в некоторых ситуациях он может послать несколько токенов. Эти дополнительные токены безопасности называются *вспомогательными* (supporting) и обычно содержат дополнительную информацию о клиентах.

Набор утверждений в технологии WCF представлен в виде абстрактного класса `System.IdentityModel.Claims.ClaimSet`, который содержит список утверждений вместе со ссылкой на их издателя.

```
[DataContract(Namespace = "http://schemas.xmlsoap.org/ws/2005/05/identity")]
public abstract class ClaimSet: IEnumerable <Claim>
{
    public abstract IEnumerable <Claim>
         FindClaims(string claimType, string right);
    public abstract IEnumerator <Claim> GetEnumerator();
    public abstract int Count { get; }
    public abstract ClaimSet Issuer { get; }
    public abstract Claim this[int index] { get; }
}
```

Три конкретные реализации этого класса поставляются как часть технологии WCF. Один из них, класс `System.IdentityModel.Claims.DefaultClaimSet`, предназначен для хранения любых утверждений, используемых для добавления или отправки дополнительных утверждений службе. Два других предназначены для превращения сертификатов X509 и токенов системы Windows в утверждения. Они называются `System.IdentityModel.Claims.X509CertificateClaimSet` и `System.IdentityModel.Claims.WindowsClaimSet` соответственно.

Как видим, основная реализация этого класса содержит разные методы перечисления или поиска существующих утверждений в наборе утверждений, а также предоставления дополнительной информации об издателе утверждений.

Все наборы утверждений, генерируемых платформой WCF, доступны во время выполнения операции благодаря классу `System.ServiceModel.ServiceSecurityContext`, который, в свою очередь, доступен для текущего контекста `OperationContext`.

```
ServiceSecurityContext securityContext
= OperationContext.Current.ServiceSecurityContext;
```

Контекст безопасности доступен для любого кода авторизации, принимающего участие в выполнении операции. Этот механизм будет рассмотрен подробнее в разделе, посвященном авторизации, а пока основное внимание уделим способу, которым этот класс запрашивает утверждения, генерируемые платформой WCF после аутентификации.

```
ServiceSecurityContext securityContext =
OperationContext.Current.ServiceSecurityContext;

IEnumerable <Claim> claims = securityContext.
AuthorizationContext.ClaimSets[0].FindClaims
                (ClaimTypes.Name, Rights.Identity);
foreach (Claim claim in claims)
{
  string name = claim.Resource as string;
  Console.WriteLine("Your name is {0}", name);
}
```

Механизм аутентификации

Итак, мы рассмотрели, как платформа WCF аутентифицирует клиентов и превращает информацию, представленную токенами безопасности, в утверждения, которые можно передать службе. Настало время привести несколько конкретных примеров и сценариев сквозной связи, используя некоторые готовые схемы аутентификации, поставляемые вместе с технологией WCF. Некоторые из схем, рассмотренных в этом разделе, представляют собой аутентификацию имени пользователя в рамках обеспечения безопасности сообщений и/или транспорта, аутентификацию Windows для обеспечения безопасности сообщений, взаимную аутентификацию по стандарту X509 для обеспечения безопасности сообщений и, наконец, сценарий аутентификации, в котором сочетается аутентификация имени пользователя и сертификата для обеспечения безопасности сообщений с помощью вспомогательных токенов.

Аутентификация имени пользователя для обеспечения безопасности сообщений

Аутентификацию имени пользователя часто называют *прямой аутентификацией* (direct authentication), поскольку клиент предоставляет мандат аутентификации в виде имени пользователя и пароля по запросу службы, а служба может проверить корректность или аутентифицировать мандат самостоятельно с помощью хранилища идентификационной информации без обращения к третьей стороне или брокеру аутентификации (рис. 8.1)

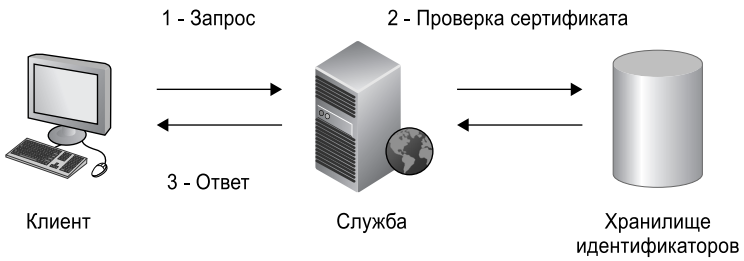


Рис. 8.1. Аутентификация имени пользователя

Основные характеристики аутентификации имени пользователя описаны ниже.

- Мандат, предъявленный службе клиентом, основан на сочетании имени пользователя и пароля. Это значит, что клиент и служба должны обменяться мандатами в режиме секретности еще до первого сеанса взаимодействия, используя внеполосный механизм.
- Служба может проверить правильность мандата клиента (имени и пароля), обратившись к хранилищу идентификационной информации. Для реализации хранилища идентификационной информации могут быть использованы разные стратегии: от простого решения, например файла, до более сложного, основанного на учетной записи пользователя системы Windows, базе данных или каталоге LDAP.
- Служба относительно проста и не требует поддержки таких функциональных возможностей, как, например, технология единого входа (single sign-on — SSO). Без технологии единого входа клиент может быть подвергнут аутентификации перед каждым вызовом службы или может кэшировать мандат в приложении.
- Клиент и служба доверяют друг другу при безопасном управлении мандатами. Если одна из сторон управляет мандатами, нарушая правила безопасности, то нет гарантии, что неправильно обработанный мандат обеспечит идентификацию клиента.

В реализации аутентификации имени пользователя в технологии WCF при обеспечении безопасности сообщений клиент передает мандат службе в рамках обмена секретными сообщениями. В этом случае сообщения защищаются с помощью сертификата X509, предоставляемого службой. Этот сертификат также работает как механизм, позволяющий клиенту аутентифицировать службу в рамках процесса взаимной аутентификации.

Мандат службы (сертификат X509) можно передать клиенту при первом обмене сообщениями или с помощью внеполосного механизма в зависимости от значения настройки `negotiateServiceCredential`.

Механизм аутентификации мандата пользователя с помощью хранилища идентификационной информации реализован на основе абстрактного класса `System.IdentityModel.Selectors.UsernamePasswordValidator` в сборке `System.IdentityModel.Selectors`.

```
public abstract class UserNamePasswordValidator
{
  public abstract void Validate(string userName, string password);
}
```

Сигнатура единственного метода, доступного в этом классе, довольно очевидна. Имя пользователя и пароль, полученные как часть мандата клиента, передаются платформой WCF этому классу для проверки. Если мандат не проходит проверку, то этот класс должен передать исключение.

Платформа WCF содержит две внутренние реализации этого класса, `System.IdentityModel.Selectors.MembershipProviderValidator`, для проверки мандатов с помощью существующего провайдера членства ASP.NET и `System.IdentityModel.Selectors.NoneUsernamePasswordValidator` для отмены проверки вообще. Эти две реализации доступны благодаря открытым членам в базовом классе `UsernamePasswordValidator`.

```
public abstract class UserNamePasswordValidator
{
  public static UserNamePasswordValidator
  CreateMembershipProviderValidator(MembershipProvider provider);
  public static UserNamePasswordValidator None { get; }
}
```

Статический метод `CreateMembershipProviderValidator` возвращает экземпляр класса `MembershipProviderValidator`, который содержит экземпляр класса `MembershipProvider` как аргумент.

Статическое свойство `None` возвращает экземпляр класса `NonUsernamePasswordValidator`.

Следующий сниппет кода демонстрирует простую реализацию этого класса, которая сравнивает имя пользователя и пароль с "вшитыми" строками.

```
public class MyUserNamePasswordValidator: UserNamePasswordValidator
{
   public override void Validate(string userName, string password)
   {
      if (userName != "joe" || password != "bar")
        throw new
      SecurityTokenValidationException("The user could not be authenticated");
   }
}
```

Если имя пользователя не "Joe" или пароль не "bar", то класс передает исключение, так что платформа WCF будет предполагать, что клиент не был аутентифицирован.

Однако по умолчанию аутентификация имени пользователя и пароля на платформе WCF осуществляется с помощью хранилища учетных записей системы Windows. Если эта настройка не была замещена конкретным элементом `<usernameAuthentication>` в поведении службы `<serviceCredentials>`, чтобы использовать класс `UsernamePasswordValidator`, то платформа WCF всегда использует хранилище учетных записей системы Windows.

```
<bindings>
  <wsHttpBinding>
    <binding name="HelloWorld">
      <security mode="Message">
        <message clientCredentialType="UserName"
                 negotiateServiceCredential="true"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
<behaviors>
  <serviceBehaviors>
    <behavior name="HelloWorld">
      <serviceCredentials>
        <userNameAuthentication userNamePasswordValidationMode=
"Windows|MembershipProvider|Custom"/>
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Настройка usernamePasswordValidationMode в элементе <usernameAuthentication> позволяет изменять режим, который платформа WCF использует для аутентификации мандата в виде имени пользователя. По умолчанию эта настройка имеет значение Windows, но она также допускает значения MembershipProvider для использования провайдера членства ASP.NET (он, в свою очередь, использует уже упоминавшийся класс MembershipProviderValidator) и Custom для задания пользовательской реализации класса UsernamePasswordValidator, если необходимо обеспечить поддержку функциональных свойств, которых нет в стандартном классе. (Примером пользовательской реализации является реализация класса MyUsernamePasswordValidation.)

В зависимости от значения, выбранного для этой настройки, пользователь должен задать дополнительные значения других связанных настроек.

В режиме проверки значение Windows поддерживает три необязательные настройки.

CacheLogonTokens. Указывает, кэшируются ли токены Windows и используются ли они повторно для одной и той же пары "имя пользователя–пароль". Аутентификация и создание токена регистрации Windows могут оказаться затратными операциями, поэтому платформа WCF кэширует токены регистрации, возвращаемые функцией LogonUser системы Win32, если эта настройка включена.

1. CachedLogonTokenLifetime. Задает максимальную продолжительность времени, в течение которого кэшируются токены Windows, если включена настройка CacheLogonToken.
2. MaxCachedLogonTokens. Задает максимальное количество токенов Windows для кэширования, если включена настройка CacheLogonToken.

```
<serviceCredentials>
            <userNameAuthentication userNamePasswordValidationMode="Windows"
                                    cacheLogonTokens="true"
                                    cachedLogonTokenLifetime="00:01:00"
                                    maxCachedLogonTokens="10"/>
</serviceCredentials>
```

Режим проверки MembershipProvider требует дополнительной настройки membershipProviderName, чтобы задать имя уже сконфигурированного провайдера членства ASP.NET. (Это имя должно существовать в списке сконфигурированных провайдеров членства.)

```
<serviceBehaviors>
  <behavior name="HelloWorld">
       <serviceCredentials>
             <userNameAuthentication userNamePasswordValidationMode=
                  "MembershipProvider"
                  membershipProviderName="MyProvider"/>
       </serviceCredentials>
  </behavior>
</serviceBehaviors>
<system.web>
    <membership>
      <providers>
         <add name="MyProvider" type="..."/>
      </providers>
    </membership>
</system.web>
```

В заключение режим проверки Custom требует настройки customUsernamePasswordValidatorType, чтобы задать тип платформы .NET для пользовательской реализации.

```
<behavior name="HelloWorld">
  <serviceCredentials>
    <userNameAuthentication userNamePasswordValidationMode="Custom"
  customUserNamePasswordValidatorType=
"MyCustomUsernamePasswordValidator, MyAssembly"/>
  </serviceCredentials>
</behavior>
```

Сертификат X509, который используется для того, чтобы защитить сеанс связи, должен быть настроен с помощью элемента <serviceCertificate> в поведении <serviceCredentials>, как показано ниже.

```
<behavior name="HelloWorld">
  <serviceCredentials>
    <serviceCertificate findValue="CN=WCFServer" storeLocation="LocalMachine"
storeName="My" x509FindType="FindBySubjectDistinguishedName"/>
    <userNameAuthentication userNamePasswordValidationMode="Windows"/>
  </serviceCredentials>
</behavior>
```

При ссылке на сертификат X509 в разделе конфигурации или в объектной модели часто необходимо задавать разные настройки (табл. 8.3), с помощью которых платформа WCF будет определять местоположение сертификата в хранилище сертификатов Windows.

Таблица 8.3. Настройки для поиска сертификата

Настройка	Описание
StoreLocation	Задает местоположение хранилища сертификатов. Возможными значениями для этой настройки являются LocalMachine, если хранилище находится на локальном компьютере, или CurrentUser — для хранилища, используемого текущим пользователем
StoreName	Задает имя для хранилища сертификатов. Чаще всего используются значения My для персональных сертификатов и TrustedPeople для сертификатов, связанных с доверенными лицами или ресурсами
X509FindType	Задает способ, с помощью которого платформа WCF находит сертификат, используя значение FindValue. Чаще всего используются значения FindBySubjectName для поиска сертификатов по атрибуту SubjectName, указанному в сертификате, или FindBySubjectDistinguishedName для осуществления более сложного поиска с помощью атрибута SubjectDistinguishedName
FindValue	Задает значение, которое платформа WCF использует для поиска сертификата

Получив первое представление о разных настройках, необходимых для выполнения сценария сквозной связи в рамках схемы аутентификации имени пользователя, рассмотрим полноценный пример службы, возвращающей утверждения аутентифицированного пользователя.

Этап 1. Создание реализации службы

Реализация службы довольно проста: она возвращает утверждения, переданные платформой WCF в контексте безопасности, как список строк (листинг 8.1).

Листинг 8.1. Реализация службы


```
namespace WCFBook.Samples
{
  [ServiceContract()]
  public interface IEchoClaims
  {
    [OperationContract]
    List <string> Echo();
  }
}
public class EchoClaims: IEchoClaims
{
  public List <string> Echo()
  {
    List <string> claims = new List <string> ();

    foreach (ClaimSet set in
          OperationContext.Current
            .ServiceSecurityContext
            .AuthorizationContext
            .ClaimSets)
   {
     foreach (Claim claim in set)
     {
       claims.Add(string.Format("{0}-{1}-{2}",
                claim.ClaimType,
                claim.Resource.ToString(),
                claim.Right));
     }
  }

  return claims;
 }
}
```

Эта служба внедрена в консольное приложение с помощью транспорта в рамках протокола `Http` (листинг 8.2).

Листинг 8.2. Главный узел службы


```
class Program
{
  static void Main(string[] args)
  {
    ServiceHost host = new ServiceHost(typeof(EchoClaims),
                  new Uri("http://localhost:8000"));
        try
        {
          host.Open();

          Console.WriteLine("Service running....");
          Console.WriteLine("Press a key to quit");
          Console.ReadKey();
        }
        finally
```

```
        {
          host.Close();
        }
    }
}
```

Этап 2. Конфигурация службы с помощью привязки `WsHttpBinding`, аутентификации имени пользователя и механизма безопасности сообщений

На этом этапе необходимо настроить главный узел службы, чтобы открыть доступ к службе `EchoService` с помощью привязки `WsHttpBinding` (в качестве службы, использующей протокол `Http`) и аутентификации имени пользователя для обеспечения безопасности сообщений.

1. Добавляем описание службы с соответствующей конечной точкой в раздел конфигурации `system.ServiceModel`.

```
<system.serviceModel>
  <services>
    <service name="WCFBook.Samples.EchoClaims"
             behaviorConfiguration="echoClaimsBehavior">
    <endpoint address="EchoClaims"
              contract="WCFBook.Samples.IEchoClaims"
              binding="wsHttpBinding"
              bindingConfiguration="echoClaimsBinding"> </endpoint>
    </service>
</services>
```

2. Настраиваем привязку `WsHttpBinding` для использования механизма защиты сообщений с аутентификацией имени пользователя.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Message">
        <message clientCredentialType="UserName"
                 negotiateServiceCredential="true"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

   Мандат службы согласовывается автоматически, поэтому клиентское приложение не обязательно инсталлировать с помощью внеполосного механизма.

3. Настраиваем поведение `serviceBehavior` с помощью сертификата службы и режима аутентификации имени пользователя.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="echoClaimsBehavior">
      <serviceCredentials>
        <serviceCertificate
           findValue="CN=WCFServer"
           storeLocation="LocalMachine"
           storeName="My"
           x509FindType="FindBySubjectDistinguishedName"/>
          <userNameAuthentication
            userNamePasswordValidationMode="Windows"/>
```

```
        </serviceCredentials>
        <serviceMetadata httpGetEnabled="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
```

В качестве первого примера мы настроили службу на аутентификацию пользователя с помощью хранилища учетных записей системы Windows (клиентское приложение должно предъявить правильную учетную запись и пароль).

Сертификат X509 `CN=WCFServer` представляет собой тестовый сертификат, созданный для данного примера с помощью утилиты `makecert.exe`. Для автоматического создания и регистрации сертификатов X509 в хранилище сертификатов системы Windows был включен сценарий `SetupCerts.bat` вместе с примерами. Прежде чем приступать к экспериментам с примерами, проверьте, работает ли этот сценарий.

Этап 3. Реализация клиентского приложения

Клиентское приложение представляет собой обычное консольное приложение, получающее имя пользователя и пароль со стандартного устройства ввода и вызывающее службу, чтобы получить утверждения пользователя (листинг 8.3).

Листинг 8.3. Реализация службы


```
static void Main(string[] args)
{
    Console.WriteLine("Enter a valid username");
    string username = Console.ReadLine();

    Console.WriteLine("Enter the password");
    string password = Console.ReadLine();

    EchoClaimsReference.EchoClaimsClient client = new
    Client.EchoClaimsReference.EchoClaimsClient();
    client.ClientCredentials.UserName.UserName = username;
    client.ClientCredentials.UserName.Password = password;

    try
    {
       string[] claims = client.Echo();
       foreach (string claim in claims)
       {
          Console.WriteLine(claim);
       }
    }
    catch (TimeoutException exception)
    {
       Console.WriteLine("Got {0}", exception.GetType());
       client.Abort();
    }
    catch (CommunicationException exception)
    {
       Console.WriteLine("Got {0}", exception.GetType());
       client.Abort();
    }
}
```

ClientCredentials — это свойство типа System.ServiceModel.ClientCredentials, принадлежащее классу System.ServiceModel.ClientBase<T>, который должен быть базовым для всех каналов WCF.

Этот класс содержит разные свойства, которые клиентское приложение может использовать, чтобы задать правильный мандат для обращения к службе. В данном примере используется мандат имени пользователя.

Этап 4. Конфигурация клиента с помощью привязки WsHttpBinding, режима безопасности сообщений и механизма аутентификации имени пользователя

На последнем этапе осуществляется настройка клиентского приложения для использования настроек, заданных для службы, т.е. конечная точка клиента должна использовать привязку WsHttpBinding с аутентификацией имени пользователя в режиме обеспечения безопасности сообщений.

1. Добавляем конечную точку клиента в раздел конфигурации system.ServiceModel.

```
<client>
      <endpoint address="http://localhost:8000/EchoClaims"
                binding="wsHttpBinding"
                bindingConfiguration="echoClaimsBinding"
                contract="EchoClaimsReference.IEchoClaims"
                name="WSHttpBinding_IEchoClaims"
                behaviorConfiguration="echoClaimsBehavior">
          <identity>
            <dns value="WCFServer"/>
          </identity>
      </endpoint>
</client>
```

   Идентификационная информация о конечной точке должна соответствовать сертификату, настроенному на стороне службы, иначе платформа WCF передаст исключение при аутентификации службы.

2. Настраиваем привязку WsHttpBinding на использование механизма безопасности сообщений с аутентификацией имени пользователя.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Message">
        <message
          clientCredentialType="UserName"
          negotiateServiceCredential="true"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

3. Поскольку мы используем тестовые сертификаты, необходимо пропустить этап проверки сертификата X509, выполняемый платформой WCF в момент аутентификации мандата службы. Если используются сертификаты X509, полученные от авторитетного источника сертификатов, этот этап является необязательным.

```
<behaviors>
  <endpointBehaviors>
    <behavior name="echoClaimsBehavior">
```

```
        <clientCredentials>
          <serviceCertificate>
            <authentication
               certificateValidationMode="None"
               revocationMode="NoCheck"/>
          </serviceCertificate>
        </clientCredentials>
      </behavior>
    </endpointBehaviors>
  </behaviors>
```

Вариант 1. Использование традиционного провайдера членства в качестве хранилища идентификационной информации

Вместо проверки мандатов пользователя с использованием хранилища учетных записей системы Windows в этом варианте описывается настройка службы на традиционного провайдера членства в виде хранилища идентификационной информации.

Для простоты мы реализовали основной провайдер членства ASP.NET. В более сложных сценариях пользователь может повторно использовать встроенные провайдеры, такие как `SqlMembershipProvider`, предназначенные для проверки мандатов с помощью базы данных SQL.

Этап 1. Реализация традиционного провайдера членства ASP.NET

Любой провайдер членства должен быть производным от базового класса `System.Web.MembershipProvider` и реализовывать разные методы аутентификации и управления пользователями в приложении. В технологии WCF для проверки мандата пользователя используется только метод `ValidateUser`, а остальные реализации игнорируются.

```
public class MyMembershipProvider: MembershipProvider
{
    public override bool ValidateUser(string username, string password)
    {
        if (username != "joe" ||
            password != "bar")
        {
            return false;
        }

        return true;
    }
}
```

Эта реализация является довольно простой и признает мандат, только если имя пользователя совпадает со строкой “`Joe`”, а пароль — со строкой “`bar`”. Для любого другого мандата она возвращает значение `false`, поэтому платформа WCF возвращает клиенту исключение.

Этап 2. Настройка службы на использование традиционного провайдера членства

Для настройки провайдера членства для службы необходимо выполнить два дополнительных этапа. Реализация традиционного провайдера членства должна быть зарегистрирована как корректный провайдер в разделе конфигурации `system.web`.

```
<system.web>
  <membership>
    <providers>
      <add name="MyMembershipProvider" type="Common.MyMembershipProvider, Common"/>
    </providers>
  </membership>
</system.web>
```

Служба должна сослаться на этого провайдера членства в элементе <username-Authentication>.

```
<serviceBehaviors>
  <behavior name="echoClaimsBehavior">
    <serviceCredentials>
      <serviceCertificate
         findValue="CN=WCFServer"
         storeLocation="LocalMachine"
         storeName="My"
         x509FindType="FindBySubjectDistinguishedName"/>
      <userNameAuthentication
         userNamePasswordValidationMode="MembershipProvider"
         membershipProviderName="MyMembershipProvider"/>
    </serviceCredentials>
    <serviceMetadata httpGetEnabled="true"/>
  </behavior>
</serviceBehaviors>
```

Вариант 2. Использование традиционной реализации класса `UserNamePasswordValidator`

Последний вариант основан на реализации традиционного класса UsernamePasswordValidator, предназначенного для проверки мандата пользователя с помощью обычного хранилища идентификационной информации.

Этап 1. Реализация традиционного класса `UserNamePasswordValidator`

Как показано выше, традиционный класс для проверки имени пользователя должен быть производным от базового класса UserNamePasswordValidator и обеспечивать реализацию метода Validate, как показано в листинге 8.4.

Листинг 8.4. Традиционная реализация класса `UserNamePasswordValidator`


```
public class MyUserNamePasswordValidator: UserNamePasswordValidator
{
  public override void Validate(string userName, string password)
  {
    if (userName != "joe" || password != "bar")
      throw new SecurityTokenValidationException("The user could not
      be authenticated");
  }
}
```

Реализация этого традиционного механизма проверки используется здесь практически так же, как и класс MembershipProvider, за исключением того, что он проверяет мандат, когда имя пользователя совпадает со строкой “joe”, а пароль — со строкой “bar”. Для любых других мандатов он передает исключение.

Этап 2. Настройка службы на использование класса UserNamePasswordValidator

Настройка традиционного механизма проверки в разделе <system.ServiceModel> довольно очевидна. Конфигурация службы должна ссылаться на этот тип реализации в элементе <usernameAuthentication>, а параметр userNamePasswordValidationMode должен быть установлен равным "Custom".

```
<serviceBehaviors>
  <behavior name="echoClaimsBehavior">
    <serviceCredentials>
      <serviceCertificate
         findValue="CN=WCFServer"
         storeLocation="LocalMachine"
         storeName="My"
         x509FindType="FindBySubjectDistinguishedName"/>
      <userNameAuthentication
          userNamePasswordValidationMode="Custom"
              customUserNamePasswordValidatorType=
              "Common.MyCustomUsernamePasswordValidator, Common"/>
    </serviceCredentials>
  <serviceMetadata httpGetEnabled="true"/>
</behavior>
```

Аутентификация имени пользователя при обеспечении безопасности транспорта

Аутентификация имени пользователя при обеспечении безопасности транспорта представляет собой еще один классический пример прямой аутентификации. При обращении к службе клиент предоставляет мандат в виде имени пользователя и пароля с помощью любого встроенного механизма транспортной аутентификации, например базовой аутентификации.

Затем служба может проверить или аутентифицировать мандат с помощью хранилища идентификационной информации, не обращаясь к третьей стороне или брокеру аутентификации. Кроме того, все сообщения и мандаты, которыми обмениваются клиент и служба, защищены механизмами безопасности, обеспечиваемыми транспортом, например протоколами SSL/TLS.

В случае использования протокола SSL мандат службы (сертификат X509) всегда передается клиенту при квитировании по протоколу SSL в момент установления связи еще до того, как клиент отправит службе какое-нибудь сообщение.

Первая версия технологии WCF не имела способа проверки мандатов пользователей с помощью обычного хранилища идентификационной информации — мандаты всегда проверялись с помощью хранилища учетных записей системы Windows. В версии 3.5 этот аспект был исправлен. Эта версия позволяет задавать конфигурацию userNamePasswordValidator так, чтобы изменять поведение, заданное по умолчанию, когда службы не внедрены в набор серверов IIS.

Включение традиционного расширения для проверки мандатов с использованием хранилища идентификационной информации о службах, внедренных в набор серверов IIS, можно выполнить с помощью расширений на уровне IIS, например, обычного модуля ASP.NET или традиционного канала WCF, эмулирующего работу механизма Basic Authentication. Эти альтернативы в книге не рассматриваются.

Следующий пример основан на службе, внедренной в IIS, которая использует протокол SSL для обеспечения безопасности транспорта, и механизм Basic Authentication

для аутентификации клиентов. Реализации клиента и службы точно такие же, как и при аутентификации сообщений.

Этап 1. Настройка виртуального каталога в наборе серверов IIS для протокола SSL

Запустим диспетчер Internet Information Services Manager (IIS Manager) и настроим сертификат сервера по конфигурации соединений веб-сервера, заданного по умолчанию, как показано на рис. 8.2. (Для этого можно использовать сертификат, созданный сценарием WCFServer, включенным в примеры.)

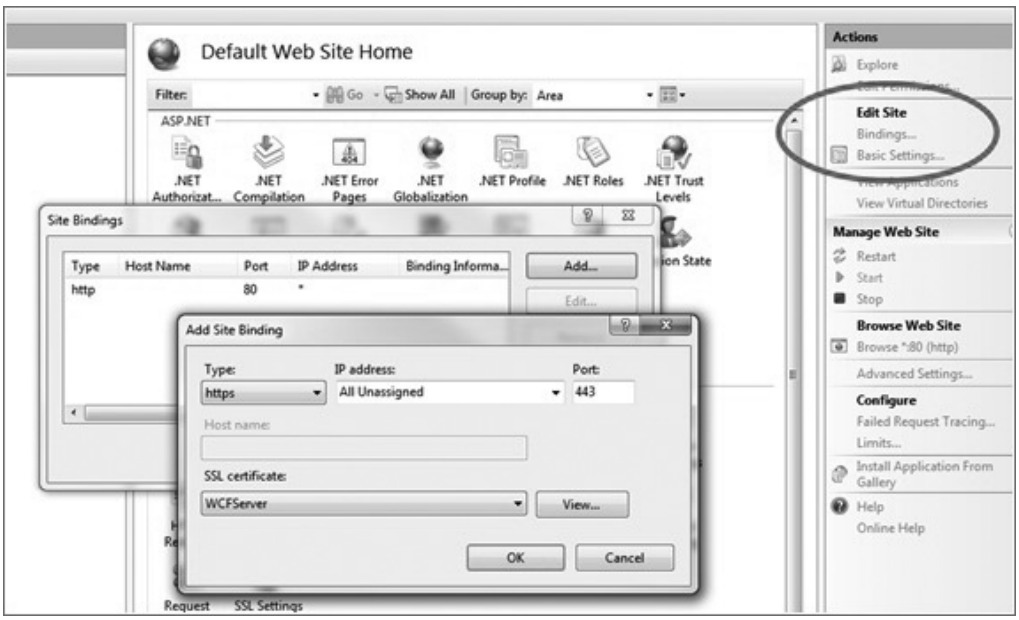


Рис. 8.2. Конфигурация виртуального каталога

Затем настроим сайт, заданный по умолчанию, в соответствии с требованиями протокола SSL (рис. 8.3).

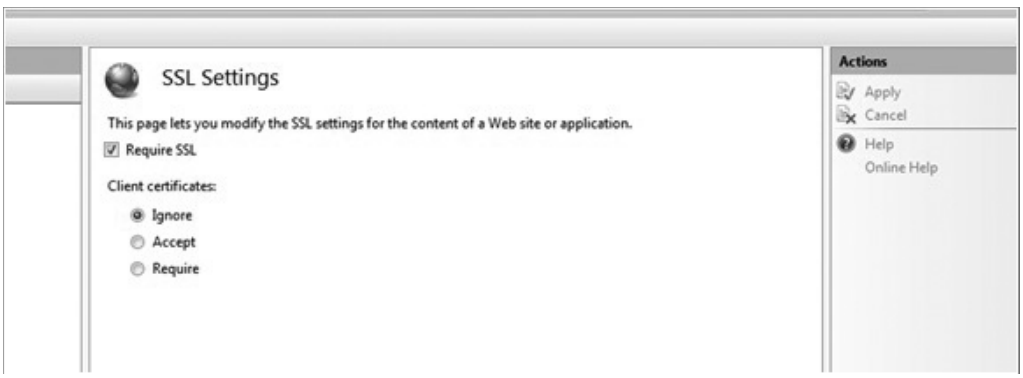


Рис. 8.3. Настройки протокола SSL

Этап 2. Создание виртуального каталога в наборе серверов IIS, настроенного с помощью механизма Basic Authentication

После настройки протокола SSL на сайт, заданный по умолчанию, можно создать виртуальный каталог, в котором будет размещена служба. Этот виртуальный каталог должен также быть настроен на механизм Basic Authentication, поскольку именно он будет использоваться для аутентификации клиентов службы (рис. 8.4).

Рис. 8.4. Настройки механизма Basic Authentication

Это можно осуществить, задав опции аутентификации в свойствах виртуального каталога. (Кроме того, необходимо отключить анонимную аутентификацию для виртуального каталога, которая задается по умолчанию при создании нового виртуального каталога.)

Этап 3. Настройка службы с помощью привязки `WsHttpBinding`, схем `Basic Authentication` и механизма безопасности транспорта

На этом этапе главный узел службы настраивается так, чтобы он стал доступным для службы `EchoService` с помощью привязки `wsHttpBinding` (в качестве службы, использующей протокол `Http`) и базовой аутентификации в рамках обеспечения безопасности транспорта.

1. Добавим описание службы вместе с соответствующей конечной точкой в раздел конфигурации `system.ServiceModel`.

```
<system.serviceModel>
  <services>
    <service name="WCFBook.Samples.EchoClaims"
             behaviorConfiguration="echoClaimsBehavior">
    <endpoint address="EchoClaims"
              contract="WCFBook.Samples.IEchoClaims"
              binding="wsHttpBinding"
              bindingConfiguration="echoClaimsBinding"> </endpoint>
   </service>
</services>
```

2. Настроим привязку `WsHttpBinding` так, чтобы она использовала механизм безопасности транспорта на основе схемы `Basic Authentication`.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Transport">
        <transport clientCredentialType="Basic"/>
```

```
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

Этап 4. Настройка клиента с помощью привязки `WsHttpBinding`, механизма безопасности транспорта и схемы `Basic Authentication`

На последнем этапе необходимо настроить клиентское приложение так, чтобы оно использовало те же самые настройки безопасности, что и служба. Иначе говоря, клиентская конечная точка должна использовать привязку `WsHttpBinding` на основе открытой аутентификации и механизма безопасности транспорта.

1. Добавим конечную точку клиента в раздел конфигурации `system.ServiceModel`.

```
<client>
       <endpoint address="http://localhost:8000/EchoClaims"
                 binding="wsHttpBinding"
                 bindingConfiguration="echoClaimsBinding"
                 contract="EchoClaimsReference.IEchoClaims"
                 name="WSHttpBinding_IEchoClaims"
                 behaviorConfiguration="echoClaimsBehavior">
          <identity>
             <dns value="WCFServer"/>
          </identity>
       </endpoint>
</client>
```

   Идентификационная информация о конечной точке должна совпадать с сертификатом, настроенным для службы; в противном случае платформа WCF передает исключение в момент аутентификации службы.

2. Настраиваем привязку `WsHttpBinding` на использование механизма безопасности транспорта на основе схемы `Basic Authentication`.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Transport">
         <transport clientCredentialType="Basic"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

3. Поскольку мы используем тестовые сертификаты, пропустим проверку корректности сертификата X509, выполняемую платформой WCF в момент аутентификации мандата службы. Этот этап можно не выполнять, если используют сертификаты X509, полученные от авторитетного источника сертификатов.

```
<behaviors>
  <endpointBehaviors>
    <behavior name="echoClaimsBehavior">
      <clientCredentials>
        <serviceCertificate>
           <authentication
              certificateValidationMode="None"
```

```
                revocationMode="NoCheck"/>
            </serviceCertificate>
          </clientCredentials>
        </behavior>
      </endpointBehaviors>
    </behaviors>
```

Взаимная аутентификация с помощью сертификатов X509 в механизме безопасности сообщений

Взаимная аутентификация с помощью сертификатов X509 представляет собой типичный пример так называемой *брокерской аутентификации* (brokered authentication), при которой служба доверяет клиенту, основываясь на мандате, изданном третьей стороной — брокером аутентификации.

В этом сценарии клиенты предъявляют мандаты аутентификации в виде сертификатов X509, изданных *органом сертификации* (certificate authority — CA) в *инфраструктуре открытого ключа* (public key infrastructure — PKI). Клиентское приложение прилагает к запросу открытый ключ клиентского сертификата X509 (или ссылку на этот сертификат) и цифровую подпись сообщений с помощью закрытого ключа клиента. Получив сообщение, служба проверяет, является ли подпись сообщения корректной, используя приложенный открытый ключ, и при необходимости выполняет дополнительные проверки, чтобы убедиться, что сертификат X509, предъявленный клиентом, не просрочен и был издан авторитетным издателем сертификатов. Клиент считается аутентифицированным, если все проверки завершились успешно (рис. 8.5)

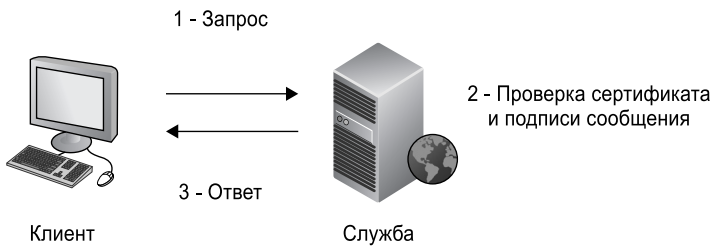


Рис. 8.5. Взаимная аутентификация с помощью сертификатов

Основные особенности взаимной аутентификации с помощью сертификатов X509 перечислены ниже.

- Мандаты, предъявленные службе клиентом, основаны на сертификатах X509. Это значит, что для издания сертификатов для любого клиента службы необходима инфраструктура PKI.
- Доверительные отношения между клиентами и службами устанавливаются при посредничестве авторитетного издателя сертификатов.

Для любого механизма аутентификации на основе безопасности сообщений характерно, что клиент передает мандат службе в ходе обмена секретными сообщениями. В этом случае сообщения защищены сертификатом X509, предъявленным службе. Этот сертификат также работает как механизм аутентификации службы клиентом в рамках процесса взаимной аутентификации. Мандат службы (сертификат X509) можно передавать клиенту в ходе первого обмена сообщениями или с помощью внеполосного механизма в соответствии с настройкой `negotiateServiceCredential`.

Стратегия аутентификации с помощью сертификатов X509 на платформе реализована в виде абстрактного класса `System.IdentityModel.Selectors.X509CertificateValidator` сборке `System.IdentityModel.Selectors`.

```
public abstract class X509CertificateValidator
{
  public abstract void Validate(X509Certificate2 certificate);
}
```

Сертификат X509, полученный как часть мандата клиента, передается платформой WCF этому классу для проверки. Если мандат не проходит проверку, класс передает исключение.

Платформа WCF содержит четыре внутренние реализации этого класса.

- Класс `System.IdentityModel.Selectors.PeerValidator` утверждает поступающие сертификаты с помощью сертификатов, инсталлированных в папке сертификатов `Trusted People`.
- Класс `System.IdentityModel.Selectors.ChainTrustValidator` подтверждает, что поступившие сертификаты используются центром сертификации сертификатов наряду с сертификатами, инсталлированными в папке сертификатов `CA Trusted`.
- Класс `System.IdentityModel.Selectors.PerOrChainTrustValidator` представляет собой комбинацию первых двух классов.
- Класс `System.IdentityModel.Selectors.NoneX509CertificateValidator` не подтверждает ничего. Доступ к этим четырем реализациям обеспечивается с помощью открытых свойств в базовом классе `X509CertificateValidator`.

```
public abstract class X509CertificateValidator
{
  public static X509CertificateValidator ChainTrust { get; }
  public static X509CertificateValidator None { get; }
  public static X509CertificateValidator PeerOrChainTrust { get; }
  public static X509CertificateValidator PeerTrust { get; }
}
```

Следующий код демонстрирует простую реализацию этого класса, которая сравнивает имя субъекта сертификата с помощью "вшитых" строк.

```
public class MyX509CertificateValidator: X509CertificateValidator
{
    public override void Validate(X509Certificate2 certificate)
    {
        if (certificate.Subject != "CN=WCFClient")
        {
          throw new SecurityTokenValidationException(
                "The X509 client certificate can not be authenticated");
        }
    }
}
```

Если имя субъекта сертификата отличается от `CN=WCFClient`, эта реализация передает исключение, а платформа WCF предполагает, что клиент не может быть аутентифицирован.

Настройки для изменения режима проверки клиентского мандата X509 находятся в элементе конфигурации <clientCertificate/authentication>, относящемся к поведению <serviceCredentials>.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="echoClaimsBehavior">
      <serviceCredentials>
        <clientCertificate>

        <authentication certificateValidationMode=
"None|PerTrust|ChainTrust|PerOrChainTrust|Custom"
                        revocationMode="NoCheck|Online|Offline"
                        />
        </clientCertificate>
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Настройка certificateValidatorMode в элементе <authentication> позволяет изменять режим, который используется платформой WCF для аутентификации мандатов с помощью сертификатов. Значения None, PerTrust, ChainTrust и PerChainTrust соответствуют внутренним реализациям класса для проверки сертификатов, рассмотренным выше. Значение Custom позволяет настроить специальную реализацию класса X509CertificateValidator, если пользователь желает обеспечить возможности, которые не поддерживаются другими опциями. (Примером специальной реализации является класс MyX509CertificatePasswordValidation.)

В зависимости от выбранного значения этой настройки пользователь должен задать дополнительные значения для других настроек, связанных с нею.

Режимы проверки PerTrust, ChainTrust и PerChainTrust имеют две дополнительные настройки.

1. TrustedStoreLocation. Задает местоположение папки доверенных сертификатов. Эта настройка имеет только два допустимых значения: CurrentUser — для хранилища, используемого текущим пользователем, и LocalMachine — для хранилища, приписанного компьютеру.
2. RevocationMode. Задает способ, с помощью которого будет выполняться проверка, не был ли сертификат аннулирован. Эта проверка сводится к поиску сертификата в списке аннулированных сертификатов. Возможные значения: Online — для проверки списка аннулированных сертификатов в режиме онлайн, Offline — для использования кэшированного списка аннулированных сертификатов и NoCheck — для отключения этой проверки.

```
<serviceCredentials>
   <clientCertificate>
      <authentication certificateValidationMode="PerTrust"
                      trustedStoreLocation="LocalMachine|CurrentUser"
                      revocationMode="NoCheck|Online|Offline"/>
   </clientCertificate>
</serviceCredentials>
```

И наконец, режим проверки Custom требует настройки customCertificate-ValidatorType для указания типа .NET, к которому относится специальная реализация механизма проверки.

```
<serviceBehaviors>
        <behavior name="echoClaimsBehavior">
          <serviceCredentials>
            <clientCertificate>
              <authentication certificateValidationMode="Custom"
                              customCertificateValidatorType=
"WCFBook.Samples.MyX509CertificateValidator, MyAssembly"
                             />
            </clientCertificate>
          </serviceCredentials>
        </behavior>
</serviceBehaviors>
```

Сертификат X509, использованный для защиты соединения, должен быть настроен с помощью элемента <serviceCertificate> в поведении <serviceCredentials>, как показано в следующем фрагменте конфигурации.

```
<serviceBehaviors>
  <behavior name="echoClaimsBehavior">
    <serviceCredentials>
      <serviceCertificate
              findValue="CN=WCFServer"
              storeLocation="LocalMachine"
              storeName="My"
              x509FindType="FindBySubjectDistinguishedName"/>
        <clientCertificate>
           <authentication certificateValidationMode="None"/>
        </clientCertificate>
      </serviceCredentials>
    <serviceMetadata httpGetEnabled="true"/>
  </behavior>
</serviceBehaviors>
```

Инструкции для обращения к сертификату, размещенные в этом разделе, совпадают с инструкциями, которые мы уже видели в сценарии аутентификации имени пользователя.

Если пользователь решит купить сертификаты у одного из хорошо известных центров сертификации, например Verisign, Thawte или Comodo, то следует убедиться, что они содержат следующие атрибуты.

```
KeyUsage:
Digital Signature, Non-Repudiation, Key Encipherment, Data Encipherment (f0)
Enhanced Key Usage: Client Authentication (1.3.6.1.5.5.7.3.2)
```

Большинство сертификатов SSL удовлетворяют этим требованиям, поэтому целесообразно воспользоваться ими, если в других сертификатах такой уверенности нет.

Настало время продемонстрировать законченный пример службы, аутентифицирующей сертификаты с помощью специального механизма проверки и возвращающей утверждения об аутентифицированном сертификате.

Этап 1. Создание реализации службы

Реализация службы в данном примере не представляет трудностей: она возвращает утверждения, переданные платформой WCF в контекст безопасности, в виде списка строк (листинг 8.5).

Листинг 8.5. Реализация службы


```
namespace WCFBook.Samples
{
  [ServiceContract()]
  public interface IEchoClaims
  {
    [OperationContract]
    List <string> Echo();
  }
}
public class EchoClaims: IEchoClaims
{
  public List <string> Echo()
  {
    List <string> claims = new List <string> ();

    foreach (ClaimSet set in
          OperationContext.Current
             .ServiceSecurityContext
             .AuthorizationContext
             .ClaimSets)
    {
      foreach (Claim claim in set)
      {
         claims.Add(string.Format("{0}-{1}-{2}",
                  claim.ClaimType,
                  claim.Resource.ToString(),
                  claim.Right));
      }
   }

   return claims;
  }
}
```

Эта служба внедрена в консольное приложение с помощью транспорта `http` (листинг 8.6).

Листинг 8.6. Реализация внедрения службы


```
class Program
{
  static void Main(string[] args)
  {
    ServiceHost host = new ServiceHost(typeof(EchoClaims),
                    new Uri("http://localhost:8000"));
    try
    {
       host.Open();
       Console.WriteLine("Service running....");
       Console.WriteLine("Press a key to quit");
       Console.ReadKey();
    }
    finally
    {
```

```
      host.Close();
    }
  }
}
```

Этап 2. Настройка службы с помощью привязки WsHttpBinding, схемы аутентификации сертификатов и механизма безопасности сообщений

На этом этапе необходимо настроить главный узел службы, чтобы открыть доступ для службы EchoService с помощью привязки wsHttpBinding (поскольку эта служба использует протокол http), схемы аутентификации сертификатов и механизма безопасности сообщений.

1. Добавим описание службы вместе с соответствующей конечной точкой в раздел конфигурации system.ServiceModel.

```
<system.serviceModel>
  <services>
    <service name="WCFBook.Samples.EchoClaims"
             behaviorConfiguration="echoClaimsBehavior">
    <endpoint address="EchoClaims"
              contract="WCFBook.Samples.IEchoClaims"
              binding="wsHttpBinding"
              bindingConfiguration="echoClaimsBinding"> </endpoint>
    </service>
</services>
```

2. Настройка привязки wsHttpBinding для использования механизма безопасности сообщений с помощью аутентификации сертификатов.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Message">
        <message clientCredentialType="Certificate"
                 negotiateServiceCredential="true"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

   Мандаты службы согласовываются автоматически, поэтому клиентское приложение необязательно инсталлировать, используя внеполосной механизм.

3. Настройка поведения serviceBehavior с помощью сертификата службы и режима аутентификации сертификата.

```
<behavior name="echoClaimsBehavior">
  <serviceCredentials>
    <serviceCertificate
            findValue="CN=WCFServer"
            storeLocation="LocalMachine"
            storeName="My"
            x509FindType="FindBySubjectDistinguishedName"/>
    <clientCertificate>
       <authentication certificateValidationMode="Custom"
                       customCertificateValidatorType=
"WCFBook.Samples.MyX509CertificateValidator, Common"
                       />
```

```
        </clientCertificate>
      </serviceCredentials>
      <serviceMetadata httpGetEnabled="true"/>
    </behavior>
```

Служба аутентифицирует сертификаты клиентов, используя специальную реализацию механизма проверки, осуществленную в типе `WCFBook.Samples.MyX509CertificateValidator`.

Сертификат X509 `CN=WCFServer` является тестовым сертификатом, созданным для данного примера с помощью утилиты `makecert.exe`. Для того чтобы автоматически создавать и регистрировать сертификаты X509 в хранилище сертификатов Windows, в примеры был включен сценарий `SetupCerts.bat`. Перед тем как приступить к экспериментам с примерами, убедитесь, что этот сценарий выполняется.

Этап 3. Реализация специального механизма проверки `X509CertificateValidator`

Как было показано ранее, специальный класс для проверки сертификатов должен быть производным от базового класса `X509CertificateValidator` и обеспечивать реализацию метода `Validate`, как показано в листинге 8.7.

Листинг 8.7. Реализация специального класса `X509CertificateValidator`


```
public override void Validate(X509Certificate2 certificate)
{
   if (certificate.Subject != "CN=WCFClient")
   {
      throw new SecurityTokenValidationException(
          "The X509 client certificate can not be authenticated");
   }
}
```

Реализация специального класса для проверки сертификатов проверяет только имя субъекта сертификата. Если имя субъекта сертификата не совпадает с `CN=WCFClient`, она передает исключение.

Этап 4. Реализация клиентского приложения

Клиентское приложение представляет собой обычное консольное приложение, вызывающее службу, чтобы получить от нее утверждения пользователя. Мандат клиента в этом случае определяется в файле конфигурации (листинг 8.8).

Листинг 8.8. Реализация клиентского приложения


```
static void Main(string[] args)
{
    EchoClaimsReference.EchoClaimsClient client = new
    Client.EchoClaimsReference.EchoClaimsClient();

    try
    {
       string[] claims = client.Echo();

       foreach (string claim in claims)
       {
```

```
            Console.WriteLine(claim);
        }
    }
    catch (TimeoutException exception)
    {
        Console.WriteLine("Got {0}", exception.GetType());
        client.Abort();
    }
    catch (CommunicationException exception)
    {
        Console.WriteLine("Got {0}", exception.GetType());
        client.Abort();
    }
}
```

Если необходимо инициализировать мандат клиента с помощью кода, то для этого удобно использовать свойство канала `ClientCredentials`, содержащее метод `SetCertificate`.

```
client.ClientCredentials.ClientCertificate.SetCertificate(
    StoreLocation.LocalMachine,
    StoreName.My,
    X509FindType.FindBySubjectDistinguishedName,
    "CN=WCFClient");
```

Этап 5. Настройка клиента с помощью привязки `WsHttpBinding`, механизма безопасности сообщений и схемы аутентификации сертификатов

На последнем этапе выполняется настройка клиентского приложения, чтобы оно использовало те же самые настройки безопасности, что и служба. Иначе говоря, конечная точка клиента должна использовать привязку `wsHttpBinding`, схему аутентификации сертификатов и механизм безопасности сообщений.

1. Добавляем конечную точку клиента в раздел конфигурации `system.ServiceModel`.

   ```
   <client>
           <endpoint address="http://localhost:8000/EchoClaims"
                     binding="wsHttpBinding"
                     bindingConfiguration="echoClaimsBinding"
                     contract="EchoClaimsReference.IEchoClaims"
                     name="WSHttpBinding_IEchoClaims"
                     behaviorConfiguration="echoClaimsBehavior">
              <identity>
                <dns value="WCFServer"/>
              </identity>
           </endpoint>
   </client>
   ```

   Идентификационная информация о конечной точке должна совпадать с сертификатом, настроенным на стороне службы; в противном случае платформа WCF передаст исключение в момент аутентификации службы.

2. Настроим привязку `WsHttpBinding` так, чтобы она использовала механизм безопасности сообщений с помощью схемы аутентификации сертификатов.

   ```
   <bindings>
     <wsHttpBinding>
   ```

```
      <binding name="echoClaimsBinding">
         <security mode="Message">
            <message
               clientCredentialType="Certificate"
               negotiateServiceCredential="true"/>
         </security>
      </binding>
    </wsHttpBinding>
</bindings>
```

3. Поскольку мы используем тестовые сертификаты, пропустим проверку корректности сертификата X509, выполняемую платформой WCF в момент аутентификации мандата службы. Этот этап можно не выполнять, если используются сертификаты X509, полученные от авторитетного источника сертификатов.

```
<behaviors>
  <endpointBehaviors>
    <behavior name="echoClaimsBehavior">
      <clientCredentials>
        <serviceCertificate>
          <authentication
            certificateValidationMode="None"
            revocationMode="NoCheck"/>
        </serviceCertificate>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
```

Аутентификация по протоколу Kerberos для обеспечения безопасности сообщений

Аутентификация по протоколу Kerberos является еще одним примером брокерской аутентификации, в котором служба доверяет клиенту, основываясь на определенном мандате (билете Kerberos), изданном третьей стороной.

В этом сценарии клиент предъявляет мандат аутентификации в виде билета Kerberos, изданного Kerberos Key Distribution Center (KDC).

Подробное описание протокола Kerberos выходит за рамки настоящей книги. Тем не менее важно понимать, что клиент должен сначала пройти аутентификацию у брокера, т.е. в центре KDC, и получить доступ к билету Kerberos, прежде чем требовать доступа к службе.

При использовании протокола Kerberos клиентское приложение запрашивает у центра KDC служебный билет, для того чтобы установить соединение с конкретной службой. В ответ на запрос клиента центр KDC создает новый сеансовый ключ и служебный билет. Билет шифруется с помощью ключа, который известен только службе (главный ключ службы). После того как клиент получит билет и сеансовый ключ, он использует их для создания токена безопасности Kerberos, который будет включен в мандат клиента как часть запроса к службе. Сеансовый ключ в этот токен не входит. Он используется только в служебном билете, который используется службой для проверки корректности токена.

Клиент защищает запрос (подписывает и, возможно, шифрует сообщение) к службе с помощью сеансового ключа и включает токен Kerberos в качестве своего мандата. Служба получает запрос, использует свой главный ключ для расшифровки служебного

билета и извлекает сеансовый ключ. Сеансовый ключ используется для верификации подписи сообщения и, возможно, его расшифровки. Затем служба возвращает ответ клиенту (хотя это не обязательно). Для обеспечения взаимной аутентификации в этом сценарии ответ должен содержать информацию, зашифрованную сеансовым ключом, чтобы доказать клиенту, что службе известен этот сеансовый ключ.

Основные особенности аутентификации по протоколу Kerberos описаны ниже.

- Протокол Kerberos обеспечивает функциональные возможности технологии SSO, позволяя клиенту аутентифицироваться только один раз в течение сеанса регистрации.
- Протокол Kerberos плотно интегрирован в операционную систему Windows, что позволяет ей обеспечивать дополнительные функциональные возможности, например заимствование прав и делегирование.
- Протокол Kerberos поддерживает взаимную аутентификацию, не требуя использовать сертификаты X509. Служба должна лишь послать ответ, содержащий данные, зашифрованные общим сеансовым ключом, чтобы доказать клиенту свою подлинность.
- Для того чтобы использовать протокол Kerberos, клиент и служба должны находиться в одном и том же лесу доменных служб, что обычно возможно только во внутренних сетевых сценариях.
- Протокол Kerberos требует, чтобы центр KDC был постоянно доступен. Если центр KDC не работает, то клиенты не могут установить доверительные отношения со службой.

Настройка `negotiateServiceCredential` в протоколе Kerberos немного отличается от других схем аутентификации, использующих сертификаты X509 в качестве мандатов службы. Если эта настройка задана равной `false`, то учетная запись, активизировавшая службу, должна ассоциироваться *идентификатором службы* (service principal name — SPN) в протоколе Kerberos. Для этого выполняемая служба должна иметь сетевую учетную запись или локальную системную учетную запись. В качестве альтернативы идентификатор SPN можно связать с учетной записью с помощью утилиты `SetSpn.exe`. В любом случае клиент должен указать правильный идентификатор SPN в идентификационной информации о конечной точке, как показано в следующем примере.

```
<client>
     <endpoint address="http://localhost:8000/EchoClaims"
                 binding="wsHttpBinding"
                 bindingConfiguration="echoClaimsBinding"
                 contract="EchoClaimsReference.IEchoClaims"
                 name="WSHttpBinding_IEchoClaims"
                 behaviorConfiguration="echoClaimsBehavior">
     <identity>
       <servicePrincipalName value="HOST/MyMachine"/>
     </identity>
  </endpoint>
</client>
```

Аутентификация по протоколу Kerberos не использует никакого высокоуровневого класса для проверки полученных мандатов, как это было при аутентификации имени пользователя и сертификата. Эта проверка может быть выполнена, но на более глубоком уровне — в стеке WCF с помощью специального класса `SecurityTokenAuthenticator`. Эта возможность в главе не рассматривается.

Изучим законченный пример службы, которая аутентифицирует клиентов с помощью протокола Kerberos и возвращает утверждения, извлеченные из контекста безопасности WCF.

Этап 1. Создание реализации службы

Реализация службы в данном примере довольно проста: она возвращает утверждения, переданные платформой WCF в контекст безопасности, в виде списка строк (листинг 8.9).

Листинг 8.9. Реализация службы


```
namespace WCFBook.Samples
{
  [ServiceContract()]
  public interface IEchoClaims
    {
      [OperationContract]
      List <string> Echo();
    }
}
public class EchoClaims: IEchoClaims
{
  public List <string> Echo()
  {
    List <string> claims = new List <string> ();

    foreach (ClaimSet set in
         OperationContext.Current
           .ServiceSecurityContext
           .AuthorizationContext
           .ClaimSets)
    {
      foreach (Claim claim in set)
      {
          claims.Add(string.Format("{0}-{1}-{2}",
                     claim.ClaimType,
                     claim.Resource.ToString(),
                     claim.Right));
      }
    }

    return claims;
  }
}
```

Служба внедрена в консольное приложение с помощью транспортного протокола `http` (листинг 8.10).

Листинг 8.10. Реализация главного узла службы


```
class Program
{
  static void Main(string[] args)
  {
```

```
    ServiceHost host = new ServiceHost(typeof(EchoClaims),
    new Uri("http://localhost:8000"));

    try
    {
      host.Open();
      Console.WriteLine("Service running....");
      Console.WriteLine("Press a key to quit");
      Console.ReadKey();
    }
    finally
    {
      host.Close();
    }
  }
}
```

Этап 2. Настройка службы с помощью привязки `WsHttpBinding`, схемы аутентификации системы Windows и механизма безопасности сообщений

На данном этапе необходимо настроить главный узел службы, чтобы сделать его доступным для службы `EchoService` с помощью привязки `wsHttpBinding` (поскольку служба использует протокол `Http`), схемы аутентификации системы Windows и механизма безопасности сообщений.

1. Добавляем описание службы в соответствующую конечную точку в раздел конфигурации `system.ServiceModel`.

```
<system.serviceModel>
  <services>
    <service name="WCFBook.Samples.EchoClaims"
             behaviorConfiguration="echoClaimsBehavior">
    <endpoint address="EchoClaims"
             contract="WCFBook.Samples.IEchoClaims"
             binding="wsHttpBinding"
             bindingConfiguration="echoClaimsBinding"> </endpoint>

    </service>
  </services>
```

2. Настраиваем привязку `wsHttpBinding` на использование механизма безопасности сообщений с помощью схемы аутентификации системы Windows.

```
<bindings>
  <wsHttpBinding>
    <binding name="echoClaimsBinding">
      <security mode="Message">
        <message clientCredentialType="Windows"
                 negotiateServiceCredential="true"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

Мандаты службы согласовываются автоматически, поэтому клиентское приложение не обязательно настраивать на службу SPN.

Этап 3. Реализация клиентского приложения

Клиентское приложение представляет собой базовое консольное приложение, вызывающее службу для получения утверждений пользователей. Мандаты клиентов согласовываются автоматически в рамках текущего сеанса регистрации в системе Windows (листинг 8.11).

Листинг 8.11. Реализация клиентского приложения


```
static void Main(string[] args)
{
    EchoClaimsReference.EchoClaimsClient client = new
    Client.EchoClaimsReference.EchoClaimsClient();

   try
   {
       string[] claims = client.Echo();
       foreach (string claim in claims)
       {
            Console.WriteLine(claim);
       }
   }
   catch (TimeoutException exception)
   {
       Console.WriteLine("Got {0}", exception.GetType());
       client.Abort();
   }
   catch (CommunicationException exception)
   {
       Console.WriteLine("Got {0}", exception.GetType());
       client.Abort();
   }
}
```

Этап 4. Настройка клиента с помощью привязки `WsHttpBinding`, механизма безопасности сообщений и схемы аутентификации системы Windows

На последнем этапе необходимо задать конфигурацию клиентского приложения с помощью тех же самых настроек, которые были использованы в конфигурации службы. Иначе говоря, необходимо настроить конечную точку клиента на использование привязки `wsHttpBinding`, схемы аутентификации системы и механизма безопасности сообщений.

1. Добавляем конечную точку в раздел конфигурации `system.ServiceModel`.

```
<client>
       <endpoint address="http://localhost:8000/EchoClaims"
                 binding="wsHttpBinding"
                 bindingConfiguration="echoClaimsBinding"
                 contract="EchoClaimsReference.IEchoClaims"
                 name="WSHttpBinding_IEchoClaims"
                 behaviorConfiguration="echoClaimsBehavior">
       </endpoint>
</client>
```

Идентификационная информация о конечной точке не нужна, поскольку она согласовывается автоматически.

2. Настраиваем привязку `wsHttpBinding` на использование механизма безопасности сообщений с помощью схемы аутентификации системы Windows.

```
<bindings>
  <wsHttpBinding>
     <binding name="echoClaimsBinding">
       <security mode="Message">
          <message
             clientCredentialType="Windows"
             negotiateServiceCredential="true"/>
       </security>
     </binding>
  </wsHttpBinding>
</bindings>
```

Преобразование утверждений и инициализация контекста безопасности

Как указывалось в разделе "Модель идентификации на основе утверждений", платформа WCF преобразует каждый успешно аутентифицированный токен в набор утверждений, которые передаются службе через контекст безопасности.

Одна из проблем, связанных с этим подходом, заключается в том, что службы часто требуют предметно-ориентированных утверждений о контексте бизнеса, к которому они относятся, например электронные адреса пользователей, лимиты продаж или синонимы пользователей. К ним также относятся технические утверждения об аутентифицированных токенах безопасности, таких как идентификаторы безопасности пользователей или имена субъектов сертификатов.

Процесс трансформации существующих утверждений или создания новых в соответствии с требованиями службы часто называют *преобразованием утверждений* (claims transformation). Поскольку платформа WCF работает только с утверждениями, довольно легко создать новый набор утверждений, используя класс `ClaimSet`. Для выполнения этого преобразования, еще до того, как утверждения будут присоединены к контексту безопасности, операции необходима лишь точка расширения.

К счастью, платформа WCF предоставляет точку расширения для этой цели и присваивает ей имя стратегии авторизации.

Стратегии авторизации выполняются платформой WCF после завершения генерации внутренних утверждений (на основе аутентифицированных токенов безопасности) и непосредственно перед выполнением операции.

Следовательно, все утверждения, связанные с мандатом клиента, также доступны в этом расширении.

Пользователь может разработать специальную стратегию авторизации в платформе WCF, реализовав интерфейс `System.IdentityModel.Policy.IAuthorizationPolicy`.

```
public interface IAuthorizationPolicy
{
   string Id { get; }
   ClaimSet Issuer { get; }
   bool Evaluate(EvaluationContext evaluationContext, ref object state);
}
```

Свойство `Id` возвращает уникальный идентификатор для этой стратегии, который обычно является идентификатором GUID. Метод `Issuer` возвращает набор утверждений, описывающий источник утверждений, созданный стратегией. В методе `Evaluate` осуществляется конкретная реализация стратегии. Этот метод получает контекст авторизации, пока он находится в процессе создания.

```
public class CustomAuthorizationPolicy: IAuthorizationPolicy
{
  string id = "Custom_" + Guid.NewGuid().ToString();

  public bool Evaluate(EvaluationContext evaluationContext, ref object state)
  {
    bool isFound = false;

    foreach (ClaimSet cs in evaluationContext.ClaimSets)
    {
       foreach (Claim claim in
cs.FindClaims(ClaimTypes.Name, Rights.PossessProperty))
       {
          if (claim.Resource.ToString()
.Equals("joe", StringComparison.InvariantCultureIgnoreCase))
         {
             isFound = true;
             break;
         }
      }
    }

    if (isFound)
    {
       evaluationContext.AddClaimSet(this,
         new DefaultClaimSet(this.Issuer,
             new Claim[] { new Claim("http://myClaimType", "I am joe",
Rights.PossessProperty) }));
    }

    return true;
  }

  public ClaimSet Issuer
  {
    get { return ClaimSet.System; }
  }

  public string Id
  {
    get { return id; }
  }
}
```

Предыдущий код демонстрирует общую реализацию стратегии авторизации, которая основана на проверке полученного контекста для поиска специфичных утверждений, содержащих имя пользователя "`Joe`". Если такое утверждение найдено, то в контекст добавляется новый набор утверждений вместе с утверждением `http://myClaimType`, специфичным для приложения.

Кроме того, стратегия авторизации представляет собой подходящее место для внедрения кода инициализации контекста безопасности службы. Настройка *специального главного объекта* (security principal) безопасности в контексте безопасности представляет собой хороший пример реализации политики авторизации.

```
public class CustomAuthorizationPolicy: IAuthorizationPolicy
{
    string id = "Custom_" + Guid.NewGuid().ToString();

    public bool Evaluate(EvaluationContext evaluationContext, ref object state)
    {
        object obj;
        if (!evaluationContext.Properties.TryGetValue("Identities", out obj))
           return false;

        IList <IIdentity> identities = obj as IList <IIdentity> ;
        if (obj == null || identities.Count <= 0)
            return false;

        evaluationContext.Properties["Principal"] =
new GenericPrincipal(identities[0], new string[]{});
        return true;
    }

    public ClaimSet Issuer
    {
        get { return ClaimSet.System; }
    }

    public string Id
    {
        get { return id; }
    }

}
```

Осталось лишь настроить стратегии авторизации в конфигурации WCF как части поведения `<serviceAuthorization>`, чтобы они вызывались каждый раз, когда в службу поступают новые запросы. Пользователь может добавить одну или несколько стратегий, и они будут вызываться в порядке их добавления.

```
<behaviors>
    <serviceBehaviors>
        <behavior name="echoClaimsBehavior">
            <serviceAuthorization>
                <authorizationPolicies>
                   <add policyType="Common.CustomAuthorizationPolicy, Common"/>
                </authorizationPolicies>
            </serviceAuthorization>
        </behavior>
    </serviceBehaviors>

</behaviors>
```

Авторизация службы

Теперь, зная, как работает механизм аутентификации на платформе WCF и как преобразовываются утверждения перед их передачей операции службы, обсудим другой интересный аспект безопасности — авторизацию.

Технология WCF предусматривает два механизма реализации авторизации в службах: простая и удобная для реализации схема, основанная на ролях пользователей, и более сложная и мощная, основанная на утверждениях. В этом разделе мы подробно опишем обе схемы и покажем, как реализовать надежную авторизацию службы.

Авторизация, основанная на ролях

Авторизация, основанная на ролях, не представляет собой ничего нового; она была известна с первой версии платформы .NET platform. Идея заключается в том, чтобы ассоциировать список ролей с пользователями, а затем, на этапе выполнения, использовать этот список для принятия решений, основанных на авторизации. Приложение может получать роли, назначенные пользователю, из разных мест и хранилищ (например, из базы данных или оснастки Azman) или определять их на основе информации о группах Windows, членом которых является пользователь.

На платформе .NET идентификационная информация о пользователе и роль, назначенная пользователю в специфичном контексте, представлены в виде интерфейсов `System.Security.IIdentity` и `System.Security.IPrincipal` соответственно.

Класс реализации `System.Security.IIdentity` обеспечивает информацию об имени пользователя (`Name`), место его аутентификации (`IsAuthenticated`) и способ его аутентификации (`AuthenticationType`).

```
public interface IIdentity
{
   string AuthenticationType { get; }
   bool IsAuthenticated { get; }
   string Name { get; }
}
```

С другой стороны, реализация класса `System.Security.IPrincipal` содержит ссылку на личность пользователя (`IIdentity`) и должна содержать метод для проверки, назначена ли пользователю специфическая роль.

```
public interface IPrincipal
{
   bool IsInRole(string role);
   IIdentity Identity { get; }
}
```

Некоторые реализации поставляются как готовые продукты, являющиеся частью платформы .NET Framework; например, в ней существуют реализации `System.Security.WindowsIdentity` и `System.Security.WindowsPrincipal` для представления пользователя системы Windows или более общие реализации, предназначенные для любых целей, в виде классов `System.Security.GenericIdentity` и `System.Security.GenericPrincipal`.

Главный объект безопасности обычно присоединяется к потоку выполнения приложения и представляет собой статическое свойство `CurrentPrincipal` в классе `System.Threading.Thread`. Одна часть приложения предназначена для инициализации этого свойства, а другая может использовать его для выполнения любого кода авторизации.

Платформа WCF работает аналогично — поведение <serviceAuthorization> управляет созданием экземпляра System.Security.IPrincipal и ассоциирует его с потоком обработки текущего запроса. Когда клиенты проходят аутентификацию системы Windows, платформа WCF пытается внедрить главный объект Windows в класс Thread.CurrentPrincipal. При выполнении аутентификации другого типа пользователь может выбирать между получением ролей от провайдера ролей ASP.NET и реализацией специальной стратегии авторизации для реализации главного объекта (в большинстве случаев используется обобщенный главный объект).

Кроме метода IsInRole, доступного в главном объекте пользователя для обеспечения безопасности, основанной на ролях, существует атрибут PermissionAttribute, позволяющий описать ролевые требования к операции службы. Если этот атрибут установлен для конкретной операции, то подсистема безопасности на платформе .NET CLR проверяет, удовлетворяет ли главный объект, присоединенный к текущему потоку, требованиям, заданным в атрибуте. Если эти требования не выполняются, то перед выполнением операции службы передается исключение SecurityException. Платформа WCF перехватывает это исключение и преобразовывает в ошибку Access Denied, которая возвращается клиенту.

Следующий сниппет кода представляет собой код авторизации, использующий атрибут PrincipalPermission, и эквивалентный вариант с использованием метода Principal.IsInRole.

```
public class Service: IService
{
   [PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
   public string DoOperation() {
   {
      return ...;
   }
}
public class Service: IService
{
   public string DoOperation() {
   {
      if (Thread.CurrentPrincipal.IsInRole("Administrators"))
      {
         return ...
      }
      else
      {
         throw new SecurityException();
      }
   }
}
```

Легко видеть, что для использования атрибута System.Security.Principal-Permission необходимо, чтобы все возможные имена групп были “зашиты” в код в процессе разработки операции службы.

Использование провайдера роли ASP.NET

Платформа WCF обеспечивает гибкое использование существующего провайдера роли ASP.NET для извлечения ролей пользователя при создании экземпляра главного объекта. Стратегия использования провайдера ролей ASP.NET очень похожа на то, что

мы видели при аутентификации членства ASP.NET. Для этого можно использовать один из встроенных провайдеров, например, провайдера роли для сервера SQL (`System.Web.Security.SqlRoleProvider`), провайдера роли для оснастки Azman (`System.Web.Security.AuthorizationStoreRoleProvider`) или провайдера ролей Windows (`System.Web.Security.WindowsTokenRoleProvider`), или разработать свой собственный провайдер, выведя его из базового класса `System.Web.Security.RoleProvider`.

Во время настройки платформы WCF на использование провайдера ролей поведение `<serviceAuthorization>` присоединяет к существующему потоку экземпляр класса `System.ServiceModel.Security.RoleProviderPrincipal`. Этот класс, по существу, создает оболочку экземпляра провайдера настраиваемой роли и передает все вызовы метода `IsInRole` методу `IsUserInRole`, принадлежащему внутреннему провайдеру ролей.

Следовательно, провайдер ролей, который будет использоваться платформой WCF, должен реализовать метод `IsUserInRole`.

```
public class MyCustomRoleProvider: RoleProvider
{
     public override string[] GetRolesForUser(string username)
     {
         if (username == "joe")
         {
            return new string[] { "administrators" };
         }
         else
         {
            return new string[] { "users" };
         }
    }

    public override bool IsUserInRole(string username, string roleName)
    {
        return GetRolesForUser(username).Contains(roleName);
    }

    #region Omitted
    #endregion

}
```

Ниже приведена конфигурация платформы WCF для использования специального провайдера роли.

```
<behaviors>
       <serviceBehaviors>
          <behavior name="echoClaimsBehavior">
             <serviceAuthorization
                principalPermissionMode="UseAspNetRoles"
                     roleProviderName="MyRoleProvider"/>
          </behavior>
       </serviceBehaviors>
</behaviors>
<system.web>
   <roleManager enabled="true">
      <providers>
        <add name="MyRoleProvider" type="Common.MyCustomRoleProvider, Common"/>
```

```
      </providers>
   </roleManager>
</system.web>
```

Использование специального главного объекта

Если пользователь решит использовать специальную реализацию главного объекта, то платформа WCF предоставит ему возможность присоединить специальный главный объект к стратегии авторизации при инициализации контекста безопасности, как указано в разделе "Преобразование утверждения и инициализации контекста безопасности".

После реализации специальной стратегии авторизации и присоединения главного объекта к контексту безопасности необходимо установить атрибут `principalPermissionMode` в поведении `serviceAuthorization` равным `Custom`, чтобы дать платформе WCF инструкцию загрузить главный объект в свойство `Thread.CurrentPrincipal`.

```
<behaviors>
       <serviceBehaviors>
          <behavior name="echoClaimsBehavior">
            <serviceAuthorization
                  principalPermissionMode="Custom">
               <authorizationPolicies>
                  <add policyType="Common.CustomAuthorizationPolicy, Common"/>
               </authorizationPolicies>
            </serviceAuthorization>
          </behavior>
       </serviceBehaviors>
</behaviors>
```

Авторизация на основе утверждений и контекст авторизации

Как известно, платформа WCF передает операциям утверждения, ассоциированные с аутентифицированными мандатами, и утверждения, созданные с помощью специальных стратегий авторизации, как часть контекста безопасности. Следовательно, есть возможность использовать всю гибкость, обеспечиваемую утверждениями, для передачи идентификационной информации или индивидуальных прав коду реализации авторизации в операциях служб.

Все наборы утверждений, сгенерированные до начала выполнения службы, доступны на платформе WCF благодаря контексту авторизации.

Платформа WCF обеспечивает доступ к контексту авторизации через класс `ServiceSecurityContext`, который является статическим по отношению к потоку и представляет собой контейнер, содержащий набор утверждений, стратегии авторизации и идентификационную информацию о текущем пользователе.

```
public class ServiceSecurityContext
{
     public AuthorizationContext AuthorizationContext { get; }
     public ReadOnlyCollection <IAuthorizationPolicy> AuthorizationPolicies { get; }
     public static ServiceSecurityContext Current { get; }
     public bool IsAnonymous { get; }
     public IIdentity PrimaryIdentity { get; }
     public WindowsIdentity WindowsIdentity { get; }
}
```

Операция службы может просто перечислить элементы набора утверждений, полученного как часть контекста авторизации, чтобы проверить, имеет ли пользователь разрешения на выполнение действия.

```
public class Service: IService
{
     public void DoAction()
     {
         bool isFound = false;

         foreach (ClaimSet cs in OperationContext
             .Current
             .ServiceSecurityContext
             .AuthorizationContext
             .ClaimSets)
        {
             foreach (Claim claim in cs.FindClaims("urn:Group",
Rights.PossessProperty))
             {
                if (claim.Resource.ToString().Equals("administrator",
StringComparison.InvariantCultureIgnoreCase))
                {
                    isFound = true;
                    break;
                }
             }
         }

        if (!isFound)
        {
            throw new SecurityException("You are not authorized");
        }
    }
}
```

В предыдущем примере операция службы искала конкретное утверждение "`urn:Group`" со значением ресурса "`administrator`". Если это утверждение не найдено, то передается исключение, поскольку пользователь не имеет разрешения на выполнение данной операции.

Менеджеры авторизации

Итак, мы видели, как использовать роль пользователя или утверждения в операции службы для реализации логики авторизации. Однако в некоторых ситуациях пользователь захочет централизовать всю логику авторизации в одном месте и применять ее к каждому поступающему запросу, не распределяя ее среди всех операций службы.

Для этой цели платформа WCF предоставляет специальную точку расширения, известную как менеджер аутентификации. Менеджер авторизации службы — это класс, производный от класса `System.ServiceModel.ServiceAuthorizationManager`, который замещает защищенный метод `CheckAccessCore` для выполнения специального кода авторизации для каждого запроса.

```
public class ServiceAuthorizationManager
{
   public virtual bool CheckAccess(OperationContext operationContext);
   public virtual bool CheckAccess(OperationContext operationContext,
```

```
ref Message message);

   protected virtual bool CheckAccessCore(OperationContext operationContext);

   protected virtual ReadOnlyCollection <IAuthorizationPolicy>
GetAuthorizationPolicies(OperationContext operationContext);
}
```

Пользователь может заместить любой виртуальный метод в этом классе; однако платформа WCF всегда вызывает метод `CheckAccessCore`.

Этот метод получает контекст операции WCF в качестве аргумента, поэтому специальная логика авторизации может проверять либо входное сообщение, либо утверждения аутентификации пользователя (или роли, если используется ролевая логика авторизации). Метод возвращает результат применения логики авторизации: "`true`", если доступ открыт, и "`false`", если в доступе отказано.

Следующий пример кода использует утверждения аутентификации для принятия решения об авторизации.

```
public class MyAuthorizationManager: ServiceAuthorizationManager
{
     protected override bool CheckAccessCore(OperationContext operationContext)
     {
         bool isFound = false;

         foreach (ClaimSet cs in OperationContext
             .Current
             .ServiceSecurityContext
             .AuthorizationContext
             .ClaimSets)
        {
             foreach (Claim claim in
cs.FindClaims("urn:Group", Rights.PossessProperty))
             {
                 if (claim.Resource.ToString().Equals("administrators",
StringComparison.InvariantCultureIgnoreCase))
                 {
                    isFound = true;
                    break;
                 }
             }
        }

        if (!isFound)
        {
           return false;
        }

        return true;
     }

}
```

Реализация специального менеджера авторизации может быть настроена на платформе WCF с помощью атрибута `serviceAuthorizationManagerType` в поведении `<serviceAuthorization>`.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="serviceBehavior">
      <serviceAuthorization
         serviceAuthorizationManagerType=
"Common.MyAuthorizationManager, Common"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Несмотря на то что использование какой-либо информации, содержащейся в сообщении (в теле SOAP), не рекомендуется из соображений эффективности, следует иметь в виду, что сообщения на платформе WCF всегда читаются один раз. Это значит, что необходимо создать копию сообщения, прежде чем читать его содержимое. Это можно сделать с помощью следующего кода, который не рекомендуется применять для больших или потоковых сообщений.

```
MessageBuffer buffer = operationContext
                .RequestContext
                .RequestMessage
                .CreateBufferedCopy(int.MaxValue);
```

9

Интегрированная аутентификация в WCF

В ЭТОЙ ГЛАВЕ...

- Основные принципы интегрированной аутентификации
- Знакомство с инфраструктурой Windows Identify Foundation (WIF)
- Реализация службы маркеров доступа с инфраструктурой WIF
- Реализация службы на основании заявлений

Эта глава посвящена исключительно интеграции между инфраструктурой Windows Identity Foundation и платформой WCF, основное внимание здесь уделяется тому, как договориться об утверждениях от службы маркеров безопасности и использовать их в целях защиты служб.

Если вы не знакомы с интегрированной аутентификацией, не волнуйтесь, поскольку ее важнейшие аспекты обсуждаются здесь в контексте работы модели безопасности платформы WCF.

Интегрированная аутентификация

Интегрированная аутентификация — это еще один пример аутентификации через посредника, где службы полагаются на третье лицо, *службу маркеров доступа* (Security Token Service — STS), для аутентификации вызывающих сторон и выдачи маркеров доступа (security token), передающих *заявления* (claim), описывающие вызывающую сторону.

Служба STS в этом контексте является мощнейшим механизмом, удовлетворяющим некоторым из следующих требований:

- Отделение служб от механизмов аутентификации и различных сертификатов, чтобы они могли сосредоточиться на авторизации или обработке корректных заявлений.
- Обеспечение интегрированной аутентификации, где клиентам, аутентифицированным в одном домене, предоставляется доступ к ресурсам или службам в другом домене при установке доверительных отношений между службами STS каждого домена.
- Преобразование заявлений в допустимый набор заявлений, ожидаемых кодом авторизации служб.

Как можно заметить, это превосходный инструмент для объединения клиентской аутентификации и избавления от множества идентификаторов, которые были созданы различными приложениями предприятия.

Что такое служба маркеров доступа (STS)

В самом общем смысле, служба STS — это реализация службы, которая, прежде всего, соответствует спецификации WS-Trust при издании маркеров доступа. Но спецификация WS-Trust —это нечто большее, поскольку она описывает контракт с четырьмя операциями: Issue, Validate, Renew и Cancel (издание, проверка, обновление и отмена). Клиенты используют эти операции для запроса новых маркеров доступа, проверки существующего маркера, возобновления маркера с истекшим сроком или отмены маркера, когда он больше не нужен.

Весь процесс использования службы, которая делегирует аутентификацию клиента службе STS, приведен на рис. 9.1.

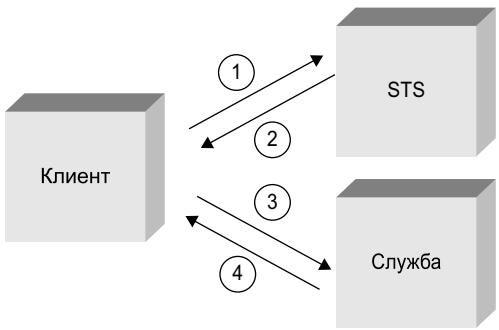


Рис. 9.1. Служба маркеров доступа

1. Приложение-клиент посылает службе STS сообщение `RequestSecurityToken` (RST), являющееся частью спецификации WS-Trust, с запросом нового маркера доступа. Это сообщение содержит клиентские сертификаты и защищается средствами безопасности, предварительно согласованными между клиентом и службой STS.
2. Служба STS получает сообщение с запросом, аутентифицирует клиента и проверяет защиту сообщения. Если проверка аутентификационной информации клиента и защиты сообщения дают позитивные результаты, служба STS использует информацию, полученную в сообщении, для создания ключа сеанса и нового маркера доступа, который включает зашифрованную версию ключа сеанса (зашифрованную с ключом, которым обладает только служба). Маркер доступа

и ключ сеанса присоединяются к сообщению `RequestSecurityTokenResponse` (RSTR) и отсылаются назад клиенту.

3. Клиент, получив сообщение RSTR, извлекает из него маркер доступа и ключ сеанса. Затем он включает маркер доступа как клиентский сертификат для сообщения запроса к службе и защищает это сообщение, используя ключ сеанса.
4. Служба получает запрос от клиента, проверяет корректность маркера доступа, использованного как клиентский сертификат, расшифровывает и извлекает из этого маркера ключ сеанса (используя тот же ключ, что и служба STS). И наконец, ключ сеанса используется для проверки защиты сообщения. Если все хорошо, выполняется операция службы, а результаты отсылаются назад клиенту.

Интегрированная аутентификация между несколькими доменами

Простая служба STS прекрасно работает, пока все участвующие стороны (клиенты, службы и служба STS) находятся в пределах той же границы доверия, которую обычно называют доменом или защищенной областью.

Все существенно усложняется, когда клиент, находящийся в одном домене, хочет использовать службу, предоставленную в другом домене. Обычно эта проблема решается за счет установки доверительных отношений между службами STS, выполняющимися в каждом домене (рис. 9.2).

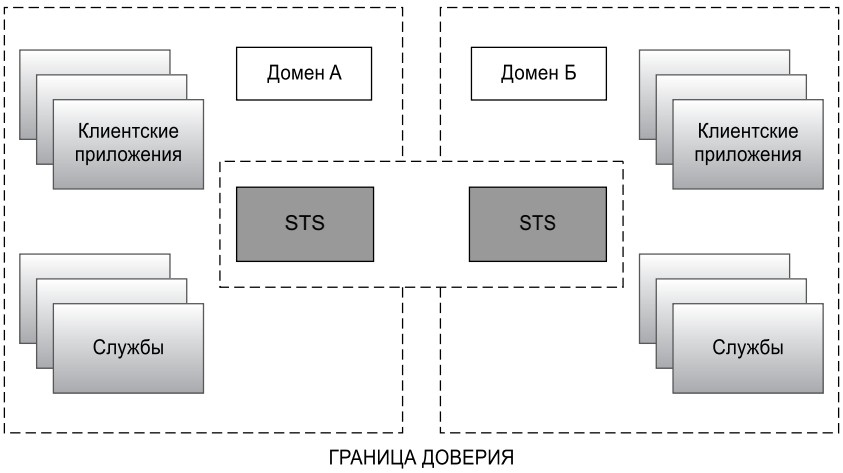


Рис. 9.2. Интегрированная аутентификация

Таким образом, клиент в домене А должен сначала получить маркер доступа от службы STS в том же домене, и использовать его позже для получения маркера доступа от службы STS, выполняющейся в домене Б. Поскольку между службами STS, работающими в обоих доменах, есть доверительные отношения, клиент должен быть в состоянии без проблем получить маркер доступа из домена В и применять его для использования любой службы, предоставляемой в этом домене.

Для клиентских приложений и служб все просто. Клиентские приложения всегда сначала аутентифицируются на службе STS, используя тот же сертификат (который обычно называют *сертификатом системы единого входа* (Single Sign On) или просто сертификатом SSO), а службы всегда получают набор заявлений, представляющих идентификационную информацию о клиентах.

За счет связывания разных служб STS тот же пример мог бы быть расширен до нескольких доменов, что создало бы доверительную сеть.

Язык разметки утверждений безопасности (SAML)

До сих пор обсуждалось, как клиентское приложение может получить от службы STS маркер доступа, который будет использован определенной службой. Как вы могли бы обратить внимание, этот маркер доступа — нечто абстрактное, он может быть представлен любым из известных маркеров доступа. Сюда относятся имена пользователя, сертификаты X509 и технология Kerberos. Но есть еще маркер доступа, который вы еще не видели, и он представляет наилучший способ передачи пользовательских заявлений в случаях с интегрированной аутентификацией. Время выйти на сцену спецификации SAML.

SAML (*язык разметки утверждений безопасности* — Security Assertions Markup Language) — независимая спецификация обмена блоками информации, связанной с безопасностью, называемых *утверждениями*, что эквивалентно *заявлению идентификации*.

Четыре следующих основных фактора поспособствовали принятию спецификации SAML для маркера доступа в случае интегрированной аутентификации.

1. Полная ориентация на язык XML гарантирует совместимость между разными платформами.
2. Ее разделы полностью расширяемы с помощью специальных атрибутов или утверждений. Как уже упоминалось, утверждение может нести информацию о заявлении идентификации.
3. Способность нести криптографическую информацию обеспечивает выполнение шифрования и работу цифровой подписи, которые обязательны для проверки защиты сообщения.
4. Способность нести цифровую подпись, созданную издателем маркера. Эта подпись чрезвычайно полезна в двух случаях: она позволяет службам удостовериться, что маркер получен с безопасной стороны, и не был изменен в пути. (В противном случае подпись не может быть подтверждена неизмененными данными маркера. Подпись не может быть также подтверждена, если содержимое маркера SAML изменилось).

В настоящее время доступны две спецификации SAML: версии 1.1 и версии 2.0.

Следующий фрагмент кода XML иллюстрирует, как выглядит маркер SAML 1.1, когда он включается в сообщение SOAP.

```
<saml:Assertion MajorVersion="1" MinorVersion="1"
AssertionID="_5d1920bc-3efa-481a-99e2-1a9469f1a128"
Issuer="http://WCFBookSTS" IssueInstant="2009-08-13T14:00:37.716Z"
xmlsn:saml="urn:oasis:names:tc:SAML:1.0:assertion">
    <saml:Conditions NotBefore="2009-08-13T14:00:37.651Z"
NotOnOrAfter="2009-08-14T00:00:37.651Z">
        < saml:AudienceRestrictionCondition>
            <saml:Audience>http://localhost:8000/EchoClaims</saml:Audience>
        </saml:AudienceRestrictionCondition>
    </saml:Conditions>
    <saml:AttributeStatement>
        <saml:Subject>
            <saml:SubjectConfirmation>
                <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
```

```
                </saml:ConfirmationMethod>
                <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsigt">
                    <!-- Зашифрованный ключ сеанса-->
                </KeyInfo>
            </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Attribute AttributeName="name"
AttributeNamespace="http://schemas.xmlsoap.org/ws/2005/05/identity/claims">
            <saml:AttributeValue>
                <!-- Значение атрибута -->
            </saml:AttributeValue>
        </saml:Attribute>
        <saml:Attribute AttributeName="AgeClaim"
AttributeNamespace="http://WCFBookSamples/2008/05">
            <saml:AttributeValue>
                <!-- Значение атрибута -- >
            </saml:AttributeValue>
        </saml:Attribute>
    </saml:AttributeStatement>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <!-- Подпись издателя -- >
    </ds:Signature>
</saml:Assertion>
```

Инфраструктура Windows Identity Foundation (WIF)

Инфраструктура *Windows Identity Foundation* (WIF), прежде вызываемая инфраструктурой *Geneva*, является новой для приложений и служб на основе заявлений, а также для реализации сценариев интегрированной защиты. Многочисленные средства этой инфраструктуры сосредоточены в основном на разработке служб маркеров доступа и их интеграции с WCF.

Модель на основе заявлений в инфраструктуре WIF

Хотя платформа WCF обладает встроенной поддержкой модели безопасности на основе заявлений, инфраструктура WIF увеличивает эти возможности, упрощая доступ к заявлениям во время выполнения и предоставляя механизм поддержки авторизации на основе заявлений, с использованием принципов ролевой авторизации, уже доступной в инфраструктуре .NET Framework. Другими словами, все классы, отправленные как часть платформы WCF, для реализации безопасности на основе заявлений, уже обсуждавшиеся в этой главе, больше не нужны, если вы решаете разрабатывать собственную модель авторизации поверх инфраструктуры WIF.

Все классы, обсуждаемые в этой главе, являются частью среды выполнения WIF. Есть также необязательный комплект SDK, предоставляющий дополнительные инструментальные средства, документацию и примеры, которые очень полезны для разработки любых приложений, предназначенных для среды выполнения WIF.

Заявления в инфраструктуре WIF представлены классом `Microsoft.IdentityModel.Claims.Claim`.

```
public class Claim
{
    public virtual string ClaimType { get; }
    public virtual string Issuer { get; }
    public virtual string OriginalIssuer { get; }
    public virtual IDictionary<string, string> Properties { get; }
    public virtual IClaimsIdentity Subject { get; }
    public virtual string Value { get; }
    public virtual string ValueType { get; }
}
```

Свойства `ClaimType`, `Issuer`и `OriginalIssuer` имеют то же назначение, что и в традиционной модели WCF. С другой стороны, значения свойств имеют небольшое отличие, поскольку они могут содержать только строковые значения вместо объектной ссылки, которая может указать на любой тип среды CLR. Таким образом, целочисленное значение 10 было бы представлено как `"10"`. Свойство `ValueType` было добавлено, оно сообщает формат значения и позволяет выяснить, как десериализировать значение заявления. Некоторыми из возможных типов значений, доступных в классе `Microsoft.IdentityModel.Claims.ClaimValueTypes`, являются:

```
public static class ClaimValueTypes
{
    public const string
        Base64Binary = "http://www.w3.org/2001/XMLSchema#base64Binary";
    public const string
        Boolean = "http://www.w3.org/2001/XMLSchema#boolean";
    public const string
        Date = "http://www.w3.org/2001/XMLSchema#date";
    public const string
        Datetime = "http://www.w3.org/2001/XMLSchema#dateTime";
    public const string
        DaytimeDuration = "http://www.w3.org/TR/2002/WD-xquery-operators-
20020816#dayTimeDuration";
    public const string
        Double = "http://www.w3.org/2001/XMLSchema#double";
    public const string
        DsaKeyValue = "http://www.w3.org/2000/09/xmldsig#DSAKeyValue";
    public const string
        HexBinary = "http://www.w3.org/2001/XMLSchema#hexBinary";
    public const string
        Integer = "http://www.w3.org/2001/XMLSchema#integer";
}
```

Свойство `Subject` ссылается на допустимый экземпляр интерфейса `Microsoft.IdentityModel.Claims.IClaimsIdentity`. Этот новый интерфейс происходит от интерфейса `System.Security.IIdentity` и представляет идентификатор темы, которой принадлежит заявление.

И наконец, если хотите включить дополнительные данные о заявлении, которые не могут быть представлены ни одним из существующих свойств, используйте словарь `Properties`. Этот словарь — общее мультимножество свойств для хранения пар “ключ/значение”.

Новые интерфейсы IClaimsIdentity и IClaimsPrincipal

Инфраструктура WIF предоставляет два новых интерфейса, представляющие клиентские идентификаторы и главные объекты, — `Microsoft.IdentityModel.Claims.IClaimsIdentity` и `Microsoft.IdentityModel.Claims.IClaimsPrincipal` соответственно (и их реализации `ClaimsIdentity` и `ClaimsPrincipal` соответственно).

Интерфейс `IClaimsIdentity`, кроме предоставления информации об идентификаторе клиента, предоставляет свойство `Claims`, позволяющее получить доступ к заявлениям, связанным с этим идентификатором. Это может быть имя, используемое при аутентификации, или способ аутентификации клиента.

```
public interface IClaimsIdentity : IIdentity
{
    ClaimCollection Claims { get; }
}
```

Поскольку есть новый интерфейс идентификатора, нужен также интерфейс главного объекта `IClaimsPrincipal` с соответствующей реализацией `ClaimsPrincipal`. Эта специфическая реализация является оболочкой экземпляра интерфейса `IClaimsIdentity` и реализует метод `IsInRole` для проверки доступных заявлений.

```
public interface IClaimsPrincipal : IPrincipal
{
    ClaimsIdentityCollection Identities { get; }
}
```

Создание активной службы STS

До появления инфраструктуры WIF создание активной службы STS с использованием платформы WCF было очень сложным делом. Вся документация и примеры были неофициальными и распространялись через различные блоги. Инфраструктура WIF заполняет важнейший недостающий фрагмент этой области, предоставляя все компоненты и классы, необходимые для реализации службы STS и применения безопасности на основе утверждений в ваших приложениях.

Основные функциональные возможности для построения специальной службы STS на платформе WCF предоставляются абстрактным классом `Microsoft.IdentityModel.SecurityTokenService.SecurityTokenService`.

Специальная служба STS наследует этот класс и обеспечивает следующие функции:

1. Конструктор, получающий экземпляр `SecurityTokenServiceConfiguration`, и используемый для настройки специфических возможностей службы STS:


```
public class MySecurityTokenService : SecurityTokenService
{
    public MySecurityTokenService( SecurityTokenServiceConfiguration
configuration )
        : base( configuration )
    {
    }
}
```

Файл Security\Authentication\Federated\STS\MySecurityTokenService.cs

2. Реализация метода `GetScope`, чтобы проверять, может ли маркер доступа быть предоставлен для службы, которую клиент хочет использовать, а также для предоставления двух соответствующих сертификатов для шифровки и подписания маркера.


```
protected override Scope GetScope( IClaimsPrincipal principal,
    RequestSecurityToken request )
{
    if (request.AppliesTo == null)
    {
        throw new InvalidRequestException("The appliesTo is null.");
    }

    if (!request.AppliesTo.Uri.Equals(new Uri("http://localhost:8000/
EchoClaims")))
    {
        Console.WriteLine("The relying party address is not valid. ");
        throw new InvalidRequestException(String.Format(
          "The relying party address is not valid. Expected value is {0},
          the actual value is {1}.",
          "http://localhost:8000/EchoClaims",
          request.AppliesTo.Uri.AbsoluteUri));
    }
    X509Certificate2 serviceCertificate = CertificateUtil.GetCertificate(
        StoreName.My,
        StoreLocation.LocalMachine,
        "CN=WCFServer");

    X509EncryptingCredentials encryptingCredentials =
        new X509EncryptingCredentials(serviceCertificate);

    Scope scope = new Scope(request.AppliesTo.Uri.AbsoluteUri,
        SecurityTokenServiceConfiguration.SigningCredentials,
        encryptingCredentials );

    return scope;
}
```

Файл Security\Authentication\Federated\STS\MySecurityTokenService.cs

Предыдущий код проверяет элемент `AppliesTo` сообщения запроса, идентифицирующий службу, которая использует маркер (`http://localhost:8000/EchoClaims`), и предоставляет сертификат с названием темы `"CN=WCFServer"` для шифрования маркера SAML и сертификата, доступного в конфигурации, чтобы подписать его.

3. Реализация метода `GetOutputClaimsIdentity`, чтобы предоставлять заявления для результирующего маркера доступа.


```
protected override IClaimsIdentity GetOutputClaimsIdentity(IClaimsPrincipal
    principal, RequestSecurityToken request, Scope scope)
{
    IClaimsIdentity callerIdentity = (IClaimsIdentity)principal.Identity;
    IClaimsIdentity outputIdentity = new ClaimsIdentity();

    Claim nameClaim = new Claim(ClaimTypes.Name, callerIdentity.Name);
```

```
        Claim ageClaim = new Claim(
            "http://WCFBookSamples/2008/05/AgeClaim",
            "25",
            ClaimValueTypes.Integer );

        outputIdentity.Claims.Add( nameClaim );
        outputIdentity.Claims.Add( ageClaim );

        return outputIdentity;
    }
```

Файл Security\Authentication\Federated\STS\MySecurityTokenService.cs

Методы `GetScope` и `GetOutputClaimsIdentity` получают в качестве аргумента допустимый экземпляр класса `IClaimsPrincipal`, который представляет аутентифицированный идентификатор вызывающей стороны. Обычно этот идентификатор используется для определения соответствующего класса, чтобы представлять вызывающую сторону. В предыдущем примере к маркеру доступа добавляются два заявления: одно для представления имени вызывающей стороны и второе жестко заданное заявление для представления возраста.

Получив готовую реализацию службы, необходимо предоставить ее одной или нескольким конечным точкам, как вы обычно сделали бы с традиционной службой WCF. В случае службы STS необходимо, предоставить один из контрактов, предлагаемых инфраструктурой WIF, который зависит, главным образом, от средств и версии спецификации WS-Trust, которую вы хотите поддерживать в службе STS.

Инфраструктура WIF имеет четыре контракта в пространстве имен `Microsoft.IdentityModel.Protocols.WSTrust`: контракт `IWSTrust13SyncContract` для предоставления конечной точки с последней версией спецификации WS-Trust (1.3), контракт `IWSTrustFeb2005SyncContract` для предоставления старшей версии спецификации (февраль 2005 года) или асинхронные версии тех же контрактов `IWSTrust13AsyncContract` и `IWSTrustFeb2005AsyncContract`.

Класс службы `WSTrustServiceContract`, который реализует четыре контракта, также доступен в том же пространстве имен. Следующий пример иллюстрирует одну из возможных конфигураций для предоставления конечной точки WS-Trust (синхронная версия спецификации WS-Trust 1.3), использующей эту реализацию службы.

```
<services>
    <service name="Microsoft.IdentityModel.Protocols.WSTrust.WSTrustServiceContract"
            behaviorConfiguration="stsBehavior">
        <endpoint address="WCFBookSTS"
            contract=
            "Microsoft.IdentityModel.Protocols.WSTrust.IWSTrust13SyncContract"
            binding="ws2007HttpBinding"
            bindingConfiguration="stsBinding"/>
    </service>
</services>
```

Вы должны определить столько конечных точек, сколько аутентификационных типов и средств спецификации WS-Trust хотите поддерживать. Определение привязки для приведенной выше конечной точки "`stsBinding`" могло бы быть следующим, если вы хотите обеспечить для этой конечной точки аутентификацию по имени пользователя.

```
<bindings>
    <ws2007HttpBinding>
        <binding name="stsBinding">
            <security mode="Message">
                <message clientCredentialType="UserName"
                    establishSecurityContext="false"
                    negotiateServiceCredential="true"/>
            </security>
        </binding>
    </ws2007HttpBinding>
</bindings>
```

Как можно заметить, для реализации службы STS с использованием инфраструктуры WIF применяются те же характеристики безопасности, которые ранее обсуждались у других служб на платформе WCF.

Последний этап реализации простой службы STS подразумевает размещение службы таким образом, чтобы она стала доступна для использования различными клиентскими приложениями. Для этого предназначен новый тип хоста службы, `Microsoft.IdentityModel.Protocols.WSTrust.WSTrustServiceHost`.

Один из конструкторов этого нового хоста службы получает как аргументы конфигурацию службы STS и базовые адреса, где служба прослушивает входящие запросы.


```
static void Main( string[] args )
{
    X509Certificate2 stsCertificate = CertificateUtil.GetCertificate(
        StoreName.My,
        StoreLocation.LocalMachine,
        "CN=WCFSTS");
    // Создать и установить конфигурацию для нашей STS
    SigningCredentials signingCreds = new X509SigningCredentials(stsCertificate);
    SecurityTokenServiceConfiguration config =
        new SecurityTokenServiceConfiguration("http://WCFBookSTS", signingCreds);

    config.SecurityTokenHandlers.AddOrReplace(new CustomUsernameTokenHandler());
    // Установить тип класса реализации STS
    config.SecurityTokenService = typeof( MySecurityTokenService );

    // Создать хост службы WS-Trust с нашей конфигурацией STS
    using ( WSTrustServiceHost host = new WSTrustServiceHost( config, new Uri(
        "http://localhost:6000" ) ) )
    {
        host.Open();
        Console.WriteLine( "WCFBookSTS started, press ENTER to stop ... " );
        Console.ReadLine();
        host.Close();
    }
}
```

Файл Security\Authentication\Federated\Program.cs

Приведенный выше код инициализирует конфигурацию STS сертификатами подписи, которые служба STS будет использовать для подписи изданных маркеров спецификации SAML, настройки специального обработчика безопасности (обсуждается далее) и наконец, создания и открытия хоста службы с использованием конфигурации и базового адреса (`http://localhost:6000`).

Обработчики маркера доступа

Обработчик маркера доступа в инфраструктуре WIF представляет новый способ внедрения специальных функций обработки маркеров. При написании обработчика маркера, вы можете добавить функциональные возможности сериализации, десериализации, аутентификации и создания маркеров специального вида.

Инфраструктура WIF обычно заменяет базовые функциональные возможности, предоставляемые платформой WCF для анализа маркеров доступа, и проверяет их, используя такие валидаторы сертификатов, как `UserNamePasswordValidator` или `X509CertificateValidator`. Эта инфраструктура все еще располагается поверх уровня служб WCF, но большая часть создаваемых вами функциональных возможностей дополняют встроенный маркер доступа WCF. Аутентификация не будут работать с этой новой архитектурой.

Базовым классом для создания нового обработчика маркера является класс `Microsoft.IdentityModel.Tokens.SecurityTokenHandler`:

```
public abstract class SecurityTokenHandler
{
    public virtual bool CanReadToken(XmlReader reader);
    public virtual SecurityToken ReadToken(XmlReader reader);
    public virtual ClaimsIdentityCollection ValidateToken(SecurityToken token);

    public virtual void WriteToken(XmlWriter writer, SecurityToken token);
    public virtual bool CanValidateToken { get; }
    public virtual bool CanWriteToken { get; }

    public abstract Type TokenType { get; }
}
```

Вы можете получить класс как производный от этого, реализовать абстрактные методы и переопределить некоторые из доступных виртуальных методов, чтобы настроить существующие обработчики маркеров доступа или создать новые.

Инфраструктура WIF обладает набором встроенных обработчиков для некоторых из известных маркеров. Сюда относится обработчик `KerberosSecurityTokenHandler` (для маркеров технологии Kerberos), обработчик `UserNameSecurityTokenHandler` (для маркеров имени пользователя), обработчик `X509SecurityTokenHandler` (для маркеров сертификата X509), обработчик `Saml11SecurityTokenHandler` (для маркеров доступа SAML 1.1) и обработчик `Saml2SecurityTokenHandle` (для маркеров доступа SAML 2.0).

Например, если хотите настроить способ авторизации маркеров имени пользователя, необходимо получить класс, производный от класса `UsernameSecurity-TokenHandler`, и предоставить реализацию метода `ValidateToken`.

```
public class CustomUsernameTokenHandler : UserNameSecurityTokenHandler
{
    public override bool CanValidateToken
    {
        get
        {
            return true;
        }
    }

    public override Microsoft.IdentityModel.Claims.ClaimsIdentityCollection
```

```
    ValidateToken(System.IdentityModel.Tokens.SecurityToken token)
    {
        UserNameSecurityToken userNameToken = token as UserNameSecurityToken;
        if (userNameToken.UserName != "joe" ||
            userNameToken.Password != "bar")
        {
            throw new SecurityTokenValidationException(
                "The user can not be authenticated");
        }

        return new ClaimsIdentityCollection(new IClaimsIdentity[] {
            new ClaimsIdentity(
            new Claim(System.IdentityModel.Claims.ClaimTypes.Name,
            userNameToken.UserName), "CustomUsernameTokenHandler")});
    }

    public override SecurityTokenHandler Clone()
    {
        return new CustomUsernameTokenHandler();
    }
}
```

В этом примере обработчик проверяет соответствие имени пользователя и пароля значениям `"Joe"` и `"bar"` соответственно. Кроме того, этот метод должен предоставить список доступных заявлений в анализируемом маркере (заявление `Name` в данном случае).

Обработчики маркеров доступа могут быть настроены либо при помощи конфигурации, либо в коде, используя класс конфигурации STS, как представлено ниже.

```
SecurityTokenServiceConfiguration config =
        new SecurityTokenServiceConfiguration("http://WCFBookSTS", signingCreds);
    config.SecurityTokenHandlers.AddOrReplace(new CustomUsernameTokenHandler());
```

Метод `AddOrReplace` заменяет встроенный обработчик `UserNameSecurityTokenHandler`, которым по умолчанию является обработчик `WindowsUserNameSecurityTokenHandler`, и проверяет маркеры, используя аутентификацию Windows.

Настройка интегрированной аутентификации в инфраструктуре WCF

Инфраструктура WCF предоставляет интеграционную привязку для поддержки случаев, когда служба ожидает маркер, изданный службой STS, который обычно является маркером SAML (хотя это может быть и любой другой). Привязка `WsFederationHttpBinding` — это исходная привязка, появившаяся в первом выпуске инфраструктуры WCF для поддержки переговоров со службой STS об издании маркера, а привязка `Ws2007FederatedHttpBinding` — это новая версия, которая поддерживает последние версии протокола WS-*.

В этом разделе вы используете привязку `Ws2007FederationHttpBinding`, чтобы настроить службу WCF, которой необходим маркер SAML версии 1.1, изданный специальной службой STS, встроенной в инфраструктуру WIF (листинг 9.1).

Если вы хотите поддерживать интегрированную аутентификацию по транспортному протоколу, отличному от HTTP, то нужно использовать специальную привязку.

Листинг 9.1. Служба на основе заявлений


```
public class EchoClaims : IEchoClaims
{
    public List<string> Echo()
    {
        List<string> claims = new List<string>();

        IClaimsPrincipal principal = Thread.CurrentPrincipal as IClaimsPrincipal;

        foreach (IClaimsIdentity identity in principal.Identities)
        {
            foreach (Claim claim in identity.Claims)
            {
                claims.Add(string.Format("{0} - {1}",
                    claim.ClaimType, claim.Value));
            }
        }
        return claims;
    }
}
```

Основное различие в том, что для доступа к заявлениям аутентификации контекст авторизации WCF больше не обязателен. Вместо этого доступный в свойстве `Thread.CurrentPrincipal` экземпляр `IClaimsPrincipal` применяется для перечисления различных экземпляров `IClaimsIdentity` и связанных с ними заявлений (поскольку вы используете встроенную поддержку, в главном объекте будет доступен только один идентификатор, первичный).

Конфигурация WCF для этой службы с использованием привязки `Ws2007FederationHttpBinding` представлена в листинге 9.2.

Листинг 9.2. Конфигурация службы


```
<services>
    <service name="WCFBook.Samples.EchoClaims"
            behaviorConfiguration="echoClaimsBehavior">
        <endpoint address="EchoClaims"
            contract="WCFBook.Samples.IEchoClaims"
            binding="ws2007FederationHttpBinding"
            bindingConfiguration="echoClaimsBinding"></endpoint>
        <endpoint address="mex"
            binding="mexHttpBinding"
            contract="IMetadataExchange" />
    </service>
</services>
<bindings>
    <ws2007FederationHttpBinding>
        <binding name="echoClaimsBinding" >
            <security mode="Message">
                <message negotiateServiceCredential="true">
                    <claimTypeRequirements>
                        <add claimType=
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
```

```
isOptional="false"/>
                        <add claimType="http://WCFBookSamples/2008/05/AgeClaim"
isOptional="false"/>
                    </claimTypeRequirements>
                </message>
            </security>
        </binding>
    </ws2007FederationHttpBinding>
</bindings>
<behaviors>
    <serviceBehaviors>
        <behavior name="echoClaimsBehavior">
            <serviceMetadata httpGetEnabled="true"/>
                <serviceCredentials>
                    <issuedTokenAuthentication
                        certificateValidationMode="None"
                        revocationMode="NoCheck">
                    <knownCertificates>
                        <add
                        findValue="CN=WCFSTS"
                        storeLocation="LocalMachine"
                        storeName="My"
                        x509FindType="FindBySubjectDistinguishedName"/>
                    </knownCertificates>
                </issuedTokenAuthentication>
                <serviceCertificate
                    findValue="CN=WCFServer"
                    storeLocation="LocalMachine"
                    storeName="My"
                    x509FindType="FindBySubjectDistinguishedName"/>
                </serviceCredentials>
            </behavior>
        </serviceBehaviors>
    </behaviors>
</system.serviceModel>
```

В привязке службы и конфигурации режима есть несколько заслуживающих внимания параметров. В конфигурации привязки элемент `<claimTypeRequirements>` указывает специальные заявления, ожидаемые службой. Эти заявления предоставляются инфраструктурой WCF в службе WS-Policy, таким образом, клиент знает, о каком заявлении следует договариваться со службой STS.

Элемент `<issuedTokenAuthentication>`, внутри элемента `<serviceCredentials>`, указывает, какая подпись X509 должна использоваться для проверки подписи маркера SAML. Этот раздел лишь инструктирует инфраструктуру WCF проверить подпись, но это все еще ответственность службы, проверять, был ли маркер издан безопасным издателем (для выполнения этой проверки инфраструктура WIF предоставляет точку расширения).

Элемент `<serviceCertificate>` определяет сертификат X509, используемый для расшифровки маркера SAML или ключ сеанса, включенный в маркер. (Это тот же сертификат, предоставленный в методе `GetScope` примера реализации службы STS.)

Подобно большинству примеров этой главы, проверка сертификата X509 была заблокирована, чтобы использовать проверочные сертификаты. Перед выпуском службы в эксплуатацию эти проверки следует возобновить.

Код для хостинга службы и поддержки безопасности на основе утверждений с инфраструктурой WIF также требует некоторых модификаций относительно традиционного способа использования.

```
static void Main(string[] args)
{
    ServiceHost host = new ServiceHost(typeof(EchoClaims),
        new Uri("http://localhost:8000"));
    try
    {
        ServiceConfiguration configuration = new
ServiceConfiguration("WCFBook.Samples.EchoClaims");
        configuration.IssuerNameRegistry = new CustomIssuerNameRegistry();

        FederatedServiceCredentials.ConfigureServiceHost(host,  configuration);

        host.Open();

        Console.WriteLine("Service running....");
        Console.WriteLine("Press a key to quit");
        Console.ReadKey();
    }
    finally
    {
        host.Close();
    }
}
```

Вызов метода `FederatedServiceCredentials.ConfigureServiceHost` запускает и инициализирует хост службы WCF со всеми расширениями, необходимыми инфраструктуре WIF, он также включает встроенные обработчики маркеров доступа.

Код также устанавливает специальный класс в свойстве конфигурации `IssuerNameRegistry`. Как упоминалось ранее, это точка расширяемости для проверки корректности издателя маркера.

Простая реализация базового класса `Microsoft.IdentityModel.Tokens.IssuerNameRegistry` представлена в листинге 9.3.

Листинг 9.3. Реализация класса `IssuerNameRegistry`


```
public class CustomIssuerNameRegistry : IssuerNameRegistry
{
    public override string GetIssuerName(System.IdentityModel.Tokens.SecurityToken
    securityToken)
    {
        X509SecurityToken token = securityToken as X509SecurityToken;
        if (token == null)
        {
            throw new SecurityTokenException("Token is not a X509 Security Token");
        }

        if (token.Certificate.SubjectName.Name != "CN=WCFSTS")
        {
            throw new SecurityTokenException("STS not supported");
        }
```

```
        return "WCFSTS";
    }
}
```

Как вы, возможно, заметили, эта реализация очень проста. Она проверяет, что маркер был подписан сертификатом X509 с именем темы "CN=WCFSTS" (только у безопасной службы STS должен быть этот сертификат закрытого ключа). Для реально работающих служб вам, вероятно, понадобятся более сложные проверки, поскольку проверка лишь имени темы, как правило, не обеспечивает надежной защиты (любой может создать достоверный сертификат с тем же именем темы).

Листинг 9.4 демонстрирует конфигурацию, подходящую для использования этой службы на стороне клиента.

Листинг 9.4. Конфигурация клиента


```
<system.serviceModel>
    <client>
        <endpoint address="http://localhost:8000/EchoClaims"
            binding="ws2007FederationHttpBinding"
            bindingConfiguration="echoClaimsBinding"
            contract="EchoClaimsReference.IEchoClaims"
            name="WSHttpBinding_IEchoClaims"
            behaviorConfiguration="echoClaimsBehavior">
            <identity>
                <dns value="WCFServer"/>
            </identity>
        </endpoint>
    </client>
    <bindings>
        <ws2007FederationHttpBinding>
            <binding name="echoClaimsBinding">
                <security mode="Message">
                    <message>
                        <claimTypeRequirements>
                            <add claimType=
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
isOptional="false"/>
                            <add claimType=
"http://WCFBookSamples/2008/05/AgeClaim"
isOptional="false"/>
                        </claimTypeRequirements>
                        <issuer address="http://localhost:6000/WCFBookSTS"
                            bindingConfiguration="stsBinding"
                            binding="ws2007HttpBinding">
                        <identity>
                            <dns value="WCFSTS"/>
                            </identity>
                        </issuer>
                        <issuerMetadata
address="http://localhost:6000/WCFBookSTS/Mex"></issuerMetadata>
                    </message>
                </security>
            </binding>
```

```
        </ws2007FederationHttpBinding>

        <ws2007HttpBinding>
            <binding name="stsBinding">
                <security mode="Message">
                    <message clientCredentialType="UserName"
                    establishSecurityContext="false"
                    negotiateServiceCredential="true"/>
                </security>
            </binding>
        </ws2007HttpBinding>
    </bindings>

    <behaviors>
        <endpointBehaviors>
            <behavior name="echoClaimsBehavior">
                <clientCredentials>
                    <serviceCertificate>
                        <defaultCertificate
                        findValue="CN=WCFSTS"
                        storeLocation="LocalMachine"
                        storeName="My"
                        x509FindType="FindBySubjectDistinguishedName"/>
                        <authentication
                        revocationMode="NoCheck"
                        certificateValidationMode="None"/>
                    </serviceCertificate>
                </clientCredentials>
            </behavior>
        </endpointBehaviors>
    </behaviors>
</system.serviceModel>
```

Основное различие с традиционной конечной точкой WCF заключается в использовании двух привязок: одна, `Ws2007FederatedBinding`, для определения подробностей связи с конечной службой, а вторая, `Ws2007HttpBinding`, для того же, но со службой STS.

Вы могли заметить, что привязка `Ws2007FederatedBinding` реплицирует ту же конфигурацию, которую вы устанавливаете на стороне сервера с дополнительным элементом `<issuer>`. Он определяет адрес издателя как ссылку привязки, используемой для установки подробностей связи.

10

Windows Azure Platform AppFabric

В ЭТОЙ ГЛАВЕ...

- Что такое Windows Azure Platform AppFabric
- Интеграция с Service Bus
- Интеграция с Access Control

Платформа Windows Azure представляет собой радикальное изменение способа создания приложений и управления ими. Эта платформа предоставляет вычислительную среду на основе "облака" Интернета, которую кто угодно может использовать для размещения приложений и хранения связанных с ними данных. В общем случае платформа заключает в себе две фундаментальные службы, которые могут использоваться любым приложением: вычислений (т.е. выполнение приложений) и хранения (т.е. хранение данных на диске).

Служба вычислений позволяет любому приложению быть запущенным в "облаке". В сущности, приложения развертываются в высоко масштабируемой среде, где они совместно используют процессорное время компьютера, доступное на различных виртуальных машинах с Windows Server. Эти экземпляры виртуальных машин распространены по всему миру в разных центрах данных Microsoft.

Служба хранения, как следует из ее названия, обеспечивает простые возможности хранения данных с использованием различных схем, таких как объекты BLOB (двоичные объекты), очереди или простые таблицы, посредством легкого в применении интерфейса прикладного программирования (API) REST на основе вызовов Http. В случае, если приложение требует более богатых возможностей запросов (например, к реляционным базам данных), имеется дополнительная служба SQL Azure, предоставляемая Microsoft для данной платформы.

В обоих случаях Windows Azure обеспечивает доступность и высокую масштабируемость, необходимые любому приложению на базе "облака" Интернета.

AppFabric расширяет платформу Windows Azure, предоставляя два общих строительных блока, Service Bus и Access Control, чтобы упростить расширение возможностей любого из приложений .NET этой платформы. В настоящее время они обеспечивают ключевые функциональные возможности по поддержке двунаправленной связи и интегрированного управления доступом для любого приложения, переносимого на платформу Windows Azure.

Основной особенностью Service Bus является маршрутизация сообщений от клиентов к вашему программному обеспечению через "облако" Windows Azure, в обход любых NAT, брандмауэров и иных препятствий, которые могут встретиться на их пути. В дополнение к маршрутизации сообщений Service Bus может также помочь в установке непосредственных связей между приложениями (рис. 10.1).

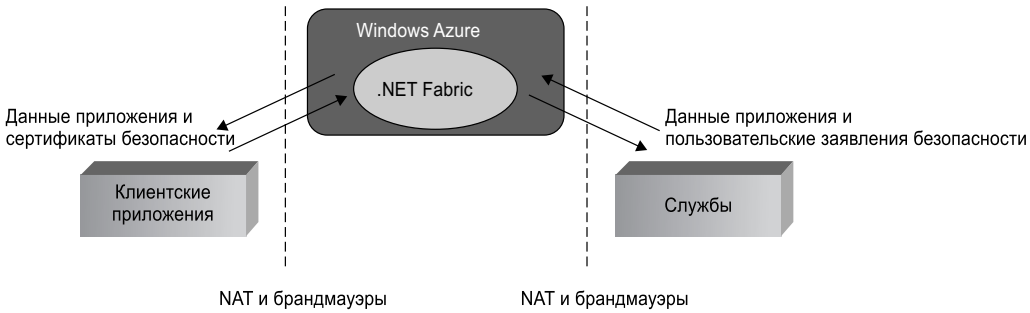


Рис. 10.1. Установка непосредственных связей между приложениями

Основной возможностью Access Control является обеспечение механизма управления доступом для приложений, работающих на конкретных локальных компьютерах или в "облаке". Это упрощает работу со встроенным механизмом аутентификации и авторизации, что позволяет сторонним приложениям выполнять проверки объектов, предоставляемых другими системами.

Хотя каждый из этих блоков доступен с использованием открытых протоколов и стандартов, таких как REST, Atom/AtomPub и SOAP, Microsoft предоставляет собственный SDK для упрощения жизни разработчиков .NET, так как он скрывает массу тонкостей протоколов связи, с которыми в противном случае разработчикам пришлось бы работать непосредственно. В рамках этого SDK вы можете воспользоваться некоторыми из новых расширений WCF. Они включают привязку ретрансляции для коммуникации с Service Bus и пользовательские диспетчеры авторизации для анализа токенов безопасности, генерируемых Access Control.

В этой главе обсуждаются теоретические основы строительных блоков AppFabric, а также многие из доступных расширений WCF, которые можно использовать для интеграции с вашими службами WCF.

Азы Service Bus и Access Control

Первым делом при работе с Service Bus и Access Control следует зайти на портал платформы Windows Azure по адресу `http://www.microsoft.com/windowsazure/`, чтобы открыть новую учетную запись и загрузить доступные инструменты и SDK.

Компоненты SDK можно загрузить со страницы, открывающейся при открытии любого из шаблонов Windows Azure в Visual Studio.

SDK для Service Bus и Access Control доступен в виде отдельной ссылки только в том случае, когда вас интересует интеграция вашего приложения с этими службами.

После инсталляции SDK становятся доступны новые сборки, обеспечивающие доступ к различным возможностям Service Bus и Access Control.

Все службы платформы Windows Azure платны, но из этого правила имеются исключения. Так, Microsoft иногда проводит акции для подписчиков MSDN; например, однажды подписчикам MSDN Premium Windows Azure были предложены восемь месяцев бесплатного использования указанных служб.

Для использования служб требуется наличие учетной записи, так что, вероятно, прежде всего вам придется зарегистрироваться. Чтобы создать учетную запись, войдите в портал с помощью вашей учетной записи Windows Live ID. После успешного входа в систему можете зарегистрироваться, чтобы получить учетную запись путем создания нового проекта. Проект в данном контексте представляет собой контейнер для всех пространств имен служб, которые будут использоваться Service Bus и Access Control.

Пространство имен службы представляет собой контейнер для некоторого набора конечных точек Service Bus и правил Access Control. Например, если вы выберете в качестве пространства имен службы для ваших конечных точек Service Bus "WCFBook", окончательным адресом для получения доступа к ним будет `sb://WCFBook.servicebus.windows.net`. Пространство имен службы должно быть глобально уникальным по всем учетным записям и иметь длину не менее шести символов.

В будущем все созданные проекты будут связаны с вашей учетной записью Windows Live. На рис. 10.2 показано, как выглядит список проектов на портале Azure.

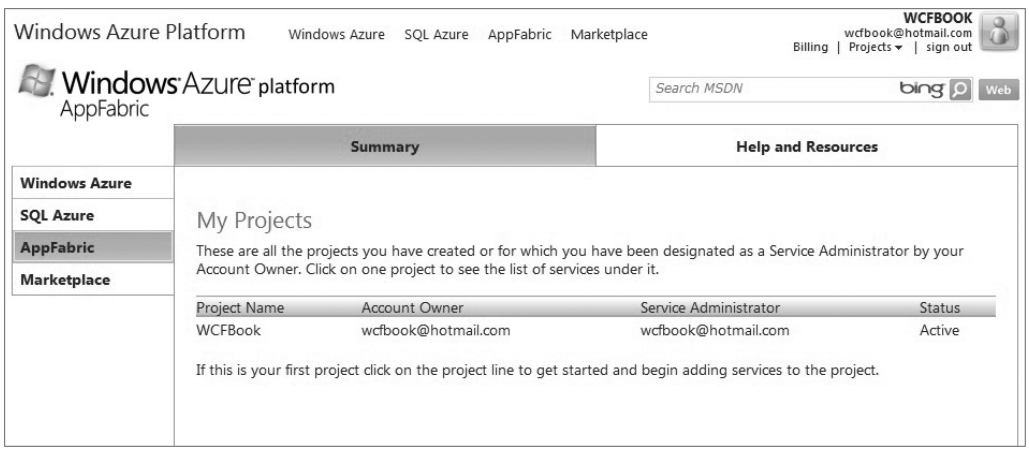


Рис. 10.2. Список проектов на портале Azure

После щелчка на одном из проектов списка вы будете перенаправлены на страницу проекта с подробной информацией о нем, где сможете добавить новые пространства имен служб. На рис. 10.3 показаны подробности проекта "WCFBook", который уже содержит связанное одноименное пространство имен службы.

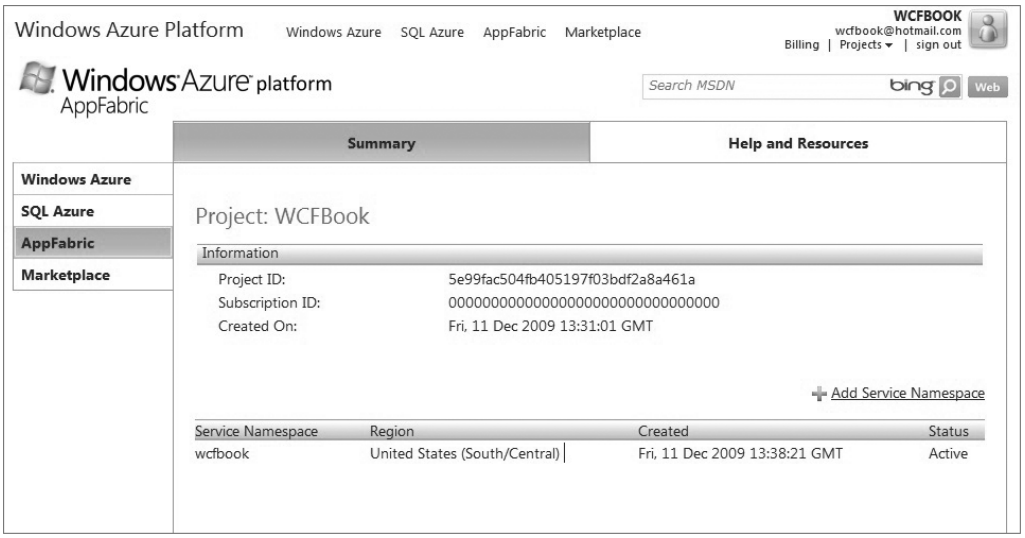


Рис. 10.3. Подробности проекта `WCFBook`

Поскольку все пространства имен служб должны быть глобально уникальными для всех имеющихся учетных записей, на экране для создания новых имен имеется кнопка для проверки доступности имени, позволяющая убедиться, что введенное вами имя не используется другой учетной записью. После того как вы создадите и свяжете пространство имен с одним из ваших проектов, инфраструктура Windows Azure выполнит необходимые шаги для резервирования и активизации этого адреса.

Чтобы перейти на страницу подробностей пространства имен службы, щелкните на нем. На этой странице можете удалить пространство имен или просмотреть все важные подробности пространства имен, такие как ключ для управления, конечные точки Service Bus и т.п. На рис. 10.4 приведена подробная информация о пространстве имен "WCFBook".

К этому моменту у вас уже должен быть установлен SDK. Кроме того, для использования возможностей Service Bus и Access Control в ваших программах и службах должна быть добавлена соответствующая новая учетная запись.

Работа с Service Bus

Service Bus на платформе Windows Azure AppFabric — это служба, работающая в центрах данных Microsoft. Ее единственная цель заключается в ретрансляции сообщений через "облако" службам, работающим за брандмауэрами или NAT. Service Bus обеспечивает безопасность конечных точек с использованием модели безопасности на базе утверждений, предоставляемой службой Access Control.

Компания Microsoft использовала название *Service Bus* из-за наличия сходств в данной архитектуре и архитектурах многих продуктов, следующих известному архитектурному шаблону ESB (Enterprise Service Bus — служебная шина предприятия).

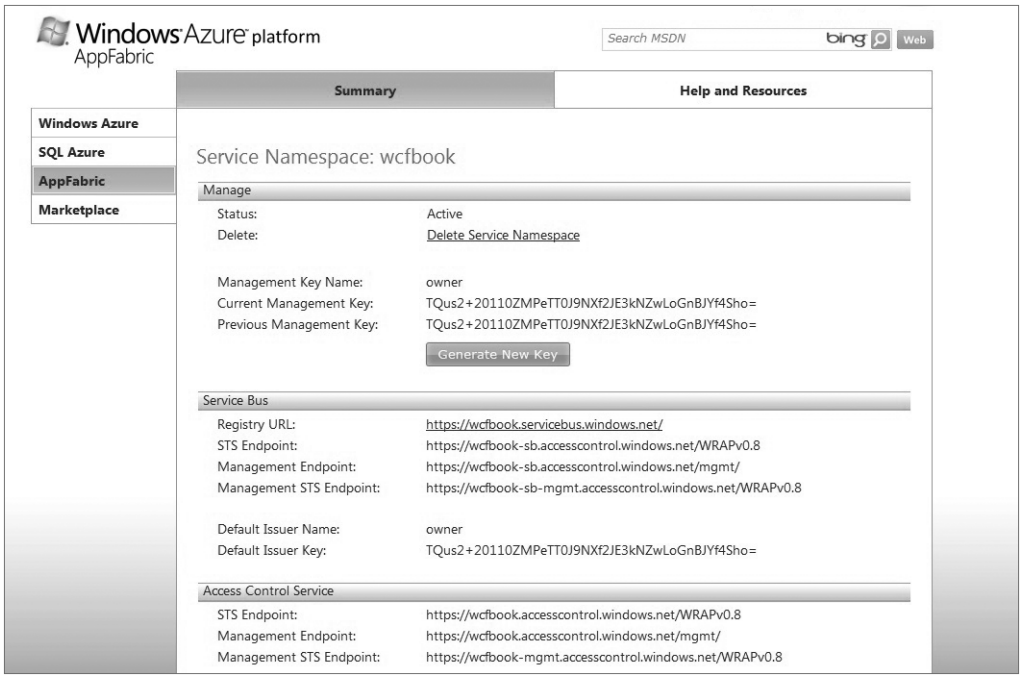


Рис. 10.4. Подробная информация о пространстве имен `WCFBook`

Шаблон ESB в реальном мире относится к модели, которая интегрирует корпоративные приложения на основе общей шины сообщений. На рис. 10.5 приведены возможные архитектуры приложения, которое использует шаблон ESB для интеграции нескольких приложений из различных платформ. Как видно на рисунке, шина вносит дополнительный уровень косвенности между клиентскими приложениями и различными службами и приложениями посредством применения архитектуры "издатель–подписчик". Другими словами, некоторые приложения подписываются на различные события или сообщения шины, в то время как другие публикуют эти события или сообщения в шине. Другие среды ESB также поддерживают службу координационного уровня, которая предоставляет процессор обработки потока для управления обменом сообщениями, составляющим коммерческий или рабочий процесс.

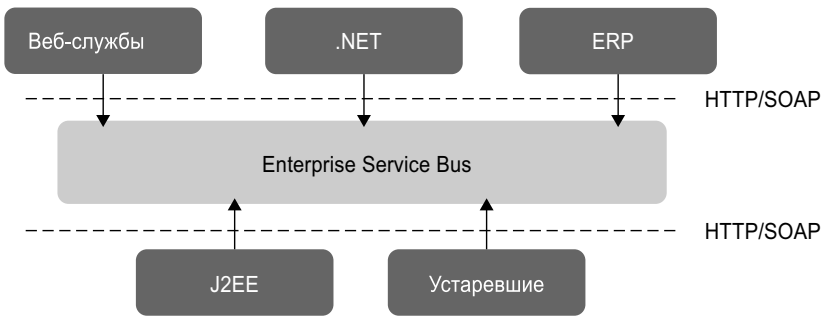


Рис. 10.5. Возможные архитектуры приложения

На сегодняшний день имеется большое количество технологий и продуктов, которые могут использоваться для реализации ESB, такие как Active Directory, UDDI, MSQM, BizTalk или WCF на платформах Windows, или многие аналогичные технологии, доступные на других платформах.

Вернемся к Service Bus, которая представляет собой часть платформы Windows Azure AppFabric. Эта служба предоставляет различные механизмы, такие как интегрированная аутентификация и управление доступом, механизм именования служб и другие, которые могут использовать приложения, интегрируемые с этой службой. Вероятно, основное отличие между данной службой Service Bus и всеми иными традиционными службами служебных шин предприятия в том, что она делает возможной интеграцию локальных приложений со службами, работающими в "облаке", или со множеством служб сторонних разработчиков, предоставляемых Microsoft или иными производителями.

Наиболее важные проблемы, с которыми пытается справиться Service Bus, представляют собой поддержку двунаправленной связи в Интернете, а также централизованной или интегрированной схемы аутентификации и авторизации.

Первая проблема — двунаправленной связи — совсем не проста для решения из-за сегодняшних реалий. Первой и более важной проблемой является нехватка адресов IPv4. Уже стало почти невозможно приобрести открытый ("белый") адрес IPv4. По этой причине большинство интернет-провайдеров (ISP), корпоративных и беспроводных сетей используют динамическое распределение IP-адресов и методы трансляции сетевых адресов (NAT). Динамические IP-адреса оказываются закрытыми в пределах своих сетей и недоступны извне.

Вторая проблема также очень важна. В настоящее время большинство корпоративных сред работают таким образом, что локальное программное обеспечение в целях безопасности полностью защищено от внешнего мира слоями брандмауэров и других защитных устройств. В большинстве случаев брандмауэры настроены таким образом, чтобы ограничить количество входящих соединений, и оставляют открытыми только порты 80 и 443 для Http- и Https-соединений. Это представляет собой большое препятствие для двунаправленного обмена данными между приложениями, который в настоящее время является весьма распространенным сценарием (например, обмен мгновенными сообщениями, обмен файлами или многопользовательские онлайн-игры).

Создание вашего первого приложения Service Bus

Как говорилось ранее, для того, чтобы использовать эту службу, вам потребуется учетная запись Service Bus, а также проекты, созданные на портале Service Bus.

Допустим, у вас уже имеется проект. Первый шаг для размещения служб состоит в их связывании с пространством имен службы. Пространство имен службы представляет собой уникальное (по всем учетным записям Service Bus) имя, выступающее в роли контейнера для множества конечных точек Service Bus. Например, если вы создаете пространство имен службы "`WCFBook`", то базовым URI для всех ваших конечных точек в данном пространстве имен будет `http://WCFBook.servicebus.windows.net`. Если вы зайдете по этому адресу с помощью браузера, то получите информацию в формате ленты новостей ATOM, которая представляет реестр Service Bus. Этот реестр содержит записи для каждой службы, содержащейся в данном пространстве имен.

Предположив, что вы уже создали пространство имен службы `WCFBook`, воспользуемся им для нашего первого примера. Он будет очень простым и будет реализовывать классическую операцию `HelloWorld` (листинг 10.1).

Листинг 10.1. Реализация службы `HelloWorld`


```
[ServiceContract]
public interface IHelloWorld
{
    [OperationContract]
    string Hello(string name);
}

public class HelloWorldService : IHelloWorld
{
    public string Hello(string name)
    {
        string message = string.Format("Hello {0}!", name);
        return message;
    }
}
```

Эта служба может быть размещена, как и любая другая обычная служба WCF. Для простоты создадим новый экземпляр класса `ServiceHost` в консольном приложении, которое считывает все детали конфигурации службы из конфигурационного файла приложения (листинг 10.2).

Листинг 10.2. Реализация приложения для размещения службы


```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Starting...");
        ServiceHost host =
            new ServiceHost(typeof(HelloWorldService));
        host.Open();
        Console.WriteLine("Press [Enter] to exit");
        Console.ReadLine();
        host.Close();
    }
}
```

Теперь настало время настроить приложение для прослушивания Service Bus. Для этого SDK предоставляет набор встроенных привязок, которые расширяют базовую инфраструктуру WCF прослушиванием “облака”. Например, `NetTcpRelayBinding` эквивалентен `NetTcpBinding`, но с иным транспортным каналом, предназначенным для прослушивания Service Bus. Привязки более подробно рассматриваются в следующем разделе.

При использовании некоторых из этих привязок необходимо также указать корректный адрес Service Bus для конечной точки. Этот адрес содержит ссылку на одно из пространств имен служб, уже созданных в веб-портале Service Bus. Например, конечный адрес для пространства имен `WCFBook` и конечной точки `HelloWorld` будет иметь вид `sb://WCFBook.servicebus.windows.net/helloWorld`.

Как можно заметить, здесь использован протокол `sb`, который представляет собой специальный протокол TCP, используемый Service Bus. Все конечные точки Service Bus, за исключением конечных точек HTTP, должны использовать протокол `sb`.

Service Bus тем или иным образом проверяет, что приложению разрешено прослушивание выбранного пространства имен службы (например, `WCFBook`). Это достигается с помощью Access Control Service. Приложение размещает службы, необходимые для предоставления учетных данных для Access Control. Access Control проверяет эти учетные данные и выдает маркер, предоставляемый Service Bus. Маркер указывает, имеет ли приложение право прослушивать, отправлять сообщения или управлять пространством имен службы. В случае, если вы являетесь владельцем пространства имен службы, этот маркер обеспечивает вам полный контроль над ним.

Учетные данные клиента могут быть предоставлены Access Control с помощью `<transportClientEndpointBehavior>` в конфигурации или эквивалентного класса `TransportClientEndpointBehavior` в коде. В настоящее время это поведение поддерживает различные типы учетных данных клиента. В табл. 10.1 приведены некоторые из них.

Таблица 10.1. Типы учетных данных клиента

Тип учетных данных	Описание
`SharedSecret`	Представляет совместно используемый ключ, автоматически назначаемый пространству имен службы при его создании
`SAML`	Маркер SAML используется для аутентификации клиента. Для успешной аутентификации Access Control должен выполнить проверку издателя маркера
`SimpleWebToken`	Для аутентификации клиента используется Simple Web Token. Это определенный тип учетных данных, более подходящий для служб RESTful, чем SAML
`Unauthenticated`	Учетные данные клиента не предоставляются

Конфигурация службы в этом примере использует тип "`SharedSecret`", который автоматически присваивается пространству имен при его создании. Учетные данные доступны на веб-портале на странице пространства имен службы в столбцах Default Issuer Name и Default Issuer Key. Вы должны использовать эти значения в атрибутах `issuerName` и `issuerSecret` конфигурации службы (листинг 10.3).

Листинг 10.3. Конфигурация службы


```
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloWorld">
        <endpoint address=
            "sb://WCFBook.servicebus.windows.net/helloWorld"
            behaviorConfiguration="clientCredentials"
            binding="netTcpRelayBinding"
            contract="IHelloWorld" />
      </service>
    </services>
    <behaviors>
      <endpointBehaviors>
        <behavior name="clientCredentials">
          <transportClientEndpointBehavior
              credentialType="SharedSecret">
            <clientCredentials>
              <sharedSecret issuerName="issuer"
                            issuerSecret="secret" />
```

```
            </clientCredentials>
          </transportClientEndpointBehavior>
        </behavior>
      </endpointBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

При первом открытии службы в консольном приложении учетные данные `SharedSecret` посылаются Access Control для получения маркера для прослушивания Service Bus.

После этого приложение, в котором размещена служба, пытается установить соединение TCP со службой ретрансляции с помощью только что полученного маркера. Если маркер не обеспечивает прослушивание Service Bus, в приложении передается исключение безопасности. В противном случае Service Bus создаст слушателя для сообщений для только что созданной службы. Таким образом, служба оказывается подключенной к "облаку".

Затем можете написать клиентское приложение, которое вызывает службу (листинг 10.4). Код этого приложения так же прост, как и код любого другого клиентского приложения, работающего со службой WCF. Все детали работы Service Bus полностью скрыты за привязкой к релею.

Листинг 10.4. Клиентское приложение


```
class Program
{
  static void Main(string[] args)
  {
    ChannelFactory < IHelloServiceBus > channelFactory =
                    new ChannelFactory < IHelloWord > ();
    IHelloWorld channel = channelFactory.CreateChannel();
    string response = channel.Hello("The WCF Book");
    Console.WriteLine(response);
    channelFactory.Close();
  }
}
```

Конфигурационный файл этого приложения очень похож на конфигурационный файл службы. Вам следует изменить конечную точку, чтобы использовать `NetTcpRelayBinding`, и использовать тот же адрес Service Bus, что и при настройке службы для прослушивания. Кроме того, необходимо сконфигурировать клиент с соответствующими учетными данными. Как это происходит и в случае служб, клиенты также должны доказать, что они имеют право отправлять сообщения на определенный адрес Service Bus. Они могут доказать это с помощью получения маркера от Access Control.

Те же данные "`SharedSecret`" используются и для настройки приложения службы. В листинге 10.5 приведена полная конфигурация клиента.

Листинг 10.5. Конфигурация клиента


```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address=
```

```
          "sb://WCFBook.servicebus.windows.net/helloWorld"
          binding="netTcpRelayBinding"
          contract="IHelloWorld"
          behaviorConfiguration="clientCredentials"
          name="helloWorld" />
    </client>
    <behaviors>
      <endpointBehaviors>
        <behavior name="clientCredentials">
          <transportClientEndpointBehavior
                credentialType="SharedSecret">
            <clientCredentials>
              <sharedSecret issuerName="issuer"
                      issuerSecret="secret" />
            </clientCredentials>
          </transportClientEndpointBehavior>
        </behavior>
      </endpointBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Теперь все готово для запуска приложения с размещенной службой с последующим приложением клиента, чтобы увидеть, как сообщение передается через Service Bus и как становится возможным обойти все сетевые препятствия на пути этого сообщения.

В следующих разделах будет рассмотрена *служба ретрансляции* (relay service), которая представляет собой ядро Service Bus, и некоторые встроенные привязки, которые любое приложение может использовать для подключения к указанной службе.

Служба ретрансляции

Служба ретрансляции представляет собой центральную систему, которая позволяет создавать приложения, способные общаться через брандмауэры и другие препятствия в сети, используя различные модели обмена сообщениями.

Эта служба поддерживает обычную однонаправленную модель, равноправную модель и модель “запрос–ответ”. Она также поддерживает две другие интересные возможности: распространение событий по Интернету с использованием архитектуры “издатель–подписчик” и двунаправленного обмена данными между приложениями, когда это возможно.

Приложение, которое использует службу ретрансляции, в основном делегирует ответственность за прослушивание транспортного уровня службе ретрансляции “облака”. Это означает, что все службы, размещенные в приложении, для обработки деталей определенного транспорта сообщений полагаются на службу ретрансляции. Эта служба будет передавать любые входящие сообщения локальным службам.

С помощью SDK вы соединяете ваши локальные службы и службы ретрансляции через любую из встроенных привязок ретрансляции. Эти привязки ретрансляции делают всю работу по замене элементов привязки транспорта некоторыми новыми элементами, интегрированными в “облаке” с Service Bus. Элементы привязки транспорта, которые поставляются как часть SDK (и интегрированы с привязками ретрансляции), устанавливают двустороннюю связь со службой ретрансляции. В ходе этого процесса выполняется также аутентификация приложения, указывается имя для прослушивания, а также тип слушателя.

Что касается адреса, выбранного для конечных точек Service Bus в службе ретрансляции, то вы должны знать, что только одна локальная служба может прослушивать некоторый конкретный адрес конечной точки, за исключением случаев использования `NetEventRelayBinding`, когда несколько служб могут прослушивать один и тот же адрес конечной точки.

Однако во всех других случаях при попытке использовать адрес, который уже используется другой службой, вы получите исключение. Более того, если этот адрес находится в области видимости URI уже используемого адреса, он также не будет работать. Например, если ваша конечная точка использует адрес `/WCFBook/MyService/HelloWorld` и имеется другая служба, которая прослушивает `/WCFBook/MyService`, то попытка использовать этот адрес потерпит неудачу. Единственное корректное решение в этом случае — когда вы совместно используете один и тот же базовый адрес для всех конечных точек (например, `/WCFBook`).

Служба ретрансляции маршрутизирует сообщения с использованием специального алгоритма, проверяющего префиксы адресов, что не обязательно означает точное соответствие. Это дает локальным службам возможность непосредственно проверить сегменты пути URI и строку запроса, что весьма полезно для служб RESTful, для выполнения пользовательской обработки.

Машине, которая пытается установить соединение со службой ретрансляции, достаточно открыть на брандмауэре лишь несколько исходящих портов, — а именно 808, 818, 819 и 828. Порт 808 используется для однонаправленного соединения TCP; порт 828 — для однонаправленного соединения TCP с SSL. Порты 818 и 819 предназначены для двунаправленных соединений TCP и некоторых других сценариев. Хорошо, что служба ретрансляции не требует открытия в брандмауэре каких-либо входящих портов или выполнения какого-либо типа отображения портов на используемых машиной устройствах NAT или маршрутизаторах.

Имеется также специальный вариант подключения на базе HTTP, который использует порты 80 (HTTP) и 443 (HTTPS), когда машина работает в среде, где все исходящие соединения заблокированы.

Пришло время детально рассмотреть привязки ретрансляции WCF и привести несколько примеров их использования.

Привязки ретрансляции WCF

Как уже говорилось ранее, WCF представляет собой основную модель программирования для работы с Service Bus на платформе .NET. SDK предоставляет набор встроенных привязок, которые облегчают интеграцию служб WCF и их потребителей через службу ретрансляции, являющуюся частью Service Bus.

В большинстве сценариев вам нужно всего лишь заменить текущую привязку WCF одной из привязок ретрансляции Service Bus. Как видно из табл. 10.2, большинство привязок ретрансляции соответствует встроенным привязкам WCF.

Таблица 10.2. Привязки ретрансляции

Привязка ретрансляции	Встроенная привязка WCF
`WebHttpRelayBinding`	`WebHttpBinding`
`BasicHttpRelayBinding`	`BasicHttpBinding`
`Ws2007HttpRelayBinding`	`Ws2007HttpBinding`
`NetTcpRelayBinding`	`NetTcpBinding`
`NetOneWayRelayBinding`	–
`NetEventRelayBinding`	–

Как видно из таблицы, только несколько привязок не имеют эквивалентных привязок WCF: `NetOneWayRelayBinding` и `NetEventRelayBinding`.

Все эти привязки в большинстве случаев работают так же, как и встроенные привязки WCF. Все они поддерживают одинаковые стандартные возможности WCF, в том числе такие, как версии сообщений, различные сценарии безопасности сообщений, надежный обмен сообщениями или модель веб-программирования.

Они просто заменяют элементы привязки транспорта WCF набором новых элементов привязки для подключения к службе ретрансляции. Эти новые элементы — `RelayedHttpBindingElement` и `RelayedHttpsBindingElement` для всех привязок на базе Http (`WebHttpRelayBinding`, `BasicHttpBinding` и `Ws2008HttpRelayBinding`); `TcpRelayTransportBindingElement` для привязок на базе TCP (`NetTcpRelayBinding`); а также `RelayedOneWayTransportBindingElement` для остальных привязок (`NetOneWayRelayBinding` и `NetEventRelayBinding`).

В следующих разделах подробно рассматриваются основные привязки ретрансляции WCF и некоторые примеры их применения.

Привязка `NetOneWayRelayBinding`

Как следует из названия данной привязки, она поддерживает только однонаправленный обмен сообщениями, что означает, что клиент не ожидает ответного сообщения от службы. Конфигурации по умолчанию для этой привязки — SOAP 1.2 с использованием TCP и бинарной кодировки. Все эти параметры могут быть настроены с помощью конфигурации привязки. С этой привязкой может использоваться только протокол `sb`.

При работе привязки с настройками по умолчанию служба WCF пытается установить исходящее безопасное соединение (TCP/SSL) со службой ретрансляции по порту 828. Когда процесс подключения к службе ретрансляции выполнен, служба WCF выполняет собственную аутентификацию посредством Access Control, указывает имя для прослушивания в службе ретрансляции и определяет тип создаваемого слушателя.

С другой стороны, когда клиент использует настройки по умолчанию, он подключается к службе ретрансляции через порт 808 (TCP) или 828 (TCP/SSL) в зависимости от конфигурации привязки. Клиенту также требуется первоначальная аутентификация с помощью Access Control, и после ее выполнения он может начать отправку односторонних сообщений службе Service Bus, которые затем ретранслируются локальной службе, прослушивающей Service Bus.

Архитектура Service Bus состоит из двух слоев. На верхнем слое находится ферма интерфейсных узлов со схемой балансировки нагрузки, которая обеспечивает необходимую масштабируемость, требуемую для службы данного типа. На слое ниже располагается серверная система именования и маршрутизации, которая в основном пересылает сообщения от клиентов интерфейсных узлов к узлам, к которым подключены службы. Это означает, что клиенты и услуги обычно подключены к разным интерфейсным узлам.

В случае, когда клиент или служба использует безопасность на уровне транспорта, с помощью SSL защищаются только сообщения между клиентом или службой и службой ретрансляции. Между узлами службы ретрансляции те же сообщения передаются в виде простого текста, так что если хотите защищать сообщения на всем их пути через службу ретрансляции, следует прибегнуть к системе безопасности на уровне сообщений.

Поскольку такая привязка поддерживает только однонаправленную передачу сообщений, все операции в контракте службы должны быть помечены как односторонние

с помощью атрибута IsOneWay. Следующий фрагмент кода иллюстрирует контракт службы, которая может использоваться с такой привязкой.

```
[ServiceContract()]
public interface IHelloWorld
{
    [OperationContract(IsOneWay = true)]
    void Hello(string name);
}
```

Реализация этой службы может быть настроена для использования NetOneWayRelayBinding с помощью следующей конфигурации.

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="clientCredentials">
        <transportClientEndpointBehavior
            credentialType="SharedSecret">
          <clientCredentials>
            <sharedSecret issuerName="ISSUER_NAME"
                          issuerSecret="ISSUER_SECRET" />
          </clientCredentials>
        </transportClientEndpointBehavior>
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <bindings>
    <netOnewayRelayBinding>
      <binding name="default" />
    </netOnewayRelayBinding>
  </bindings>
  <services>
    <service name="HelloWorldService">
      <endpoint address=""
        behaviorConfiguration="clientCredentials"
        binding="netOnewayRelayBinding"
        bindingConfiguration="default"
        name="RelayEndpoint" contract="IHelloWorld" />
    </service>
  </services>
</system.serviceModel>
```

Единственная конечная точка указывает, что для аутентификации посредством Access Control применяется совместно используемый ключ.

Клиентское приложение-потребитель этой службы должно быть настроено с такой же конфигурацией конечной точки (и также на основе NetOneWayRelayBinding).

Как уже отмечалось ранее, эта привязка по умолчанию всегда пытается подключиться к службе ретрансляции с применением TCP. Когда служба размещена в среде с ограничениями, где такая связь не может быть установлена, можете выбрать альтернативный режим, который использует традиционные порты Http и Https. SDK предоставляет специальные настройки для изменения режима связи, использующие имя ConnectivityMode. Этот параметр, как правило, принимает одно из следующих значений: TCP, Http или AutoDetect.

- `TCP` всегда пытается установить соединение со службой ретрансляции посредством TCP через порт 828.
- `Http` пытается установить соединение со службой ретрансляции посредством Http, чтобы обойти все вопросы, связанные с открытием порта 828.
- `AutoDetect` — режим подключения по умолчанию. Он определяет наилучший способ подключения путем выявления вариантов, доступных в данной сетевой среде.

Изменить режим подключения можно с помощью статического класса `ServiceBusEnvironment`, область видимости которого — уровень домена приложения. Этот класс содержит свойство `SystemConnectivity`, которому может быть присвоено любое из рассмотренных выше значений.

Приведенный ниже фрагмент кода иллюстрирует изменение приложения для использования режима подключения TCP.

```
ServiceBusEnvironment.SystemConnectivity.Mode =
        ConnectivityMode.Tcp;
ServiceHost host =
        new ServiceHost(typeof(HelloWorldService), address);
```

Эта настройка годится и для любых других привязок ретрансляции.

Привязка `NetEventRelayBinding`

С точки зрения реализации привязка `NetEventRelayBinding` практически идентична привязке `NetOneWayRelayBinding`. Обе они поддерживают одни и те же настройки по умолчанию и параметры безопасности и, по сути, одинаково взаимодействуют со службой ретрансляции.

Однако сценарий, для которого предназначено это связывание, позволяет нескольким службам WCF зарегистрироваться в одном и том же адресе Service Bus. Таким образом, генерируется архитектура типа “подписчик–издатель”, где клиент представляет собой издателя, который отправляет сообщение на определенный адрес службы в Service Bus, а служба ретрансляции рассылает это сообщение всем службам WCF, подписанным в настоящий момент на этот адрес.

Однако следует иметь в виду, что эта привязка не дает никаких гарантий по поводу доставки сообщения или порядка доставки, так что мы получаем аналог многоадресной рассылки UDP с улучшенной обработкой ошибок, поскольку служба ретрансляции для связи использует TCP.

В приведенном далее примере показаны приложение, которое регистрирует одну службу WCF для получения сообщений с использованием `NetEventRelayBinding`, и простое клиентское приложение для работы с ней.

Одновременно может быть запущено несколько экземпляров этого приложения, поскольку все они совместно используют один и тот же адрес.

В листинге 10.6 показаны контракт и реализация службы WCF. В действительности служба ничего не делает с полученным сообщением и лишь выводит его на консоль.

Листинг 10.6. Контракт и реализация службы


```
[ServiceContract()]
public interface ISubscriber
{
    [OperationContract(IsOneWay = true)]
    void ReceiveMessage(string message);
}

[ServiceBehavior()]
class SubscriberService : ISubscriber
{
    void ISubscriber.ReceiveMessage(string message)
    {
        Console.WriteLine("Receive " + message);
    }
}
```

Для простоты клиент и служба работают в одном и том же консольном приложении. Метод `Run` создает новое размещение для службы `SubscriberService`, а после этого — канал клиента для работы с ней (листинг 10.7).

Листинг 10.7. Реализация приложения


```
static void Main(string[] args)
{
    ServiceBusEnvironment.SystemConnectivity.Mode =
            ConnectivityMode.Tcp;
    Program programInstance = new Program();
    programInstance.Run();
}

private void Run()
{
    ServiceHost host =
            new ServiceHost(typeof(SubscriberService));
    host.Open();

    ChannelFactory < ISubscriberChannel > channelFactory =
        new ChannelFactory <ISubscriberChannel>
        ("RelayEndpoint");

    ISubscriberChannel channel =
        channelFactory.CreateChannel();
    channel.Open();

    Console.WriteLine("\nPress [Enter] to exit\n");

    channel.ReceiveMessage("This is a message!!!");

    channel.Close();
    channelFactory.Close();
    host.Close();
}
```

Поскольку и клиент, и служба работают в одном и том же приложении, конфигурация WCF содержит конфигурации каждого из них. Как можно видеть, `NetEventRelayBinding` был настроен и для клиента, и для службы, а аутентификация с помощью Access Control выполнялась с помощью совместно используемого ключа (листинг 10.8).

Листинг 10.8. Конфигурация клиента и службы


```
<system.serviceModel>

  <bindings>
    <netEventRelayBinding>
      <binding name="default"/>
    </netEventRelayBinding>
  </bindings>

  <client>
    <endpoint
      name="RelayEndpoint"
      contract="ISubscriber"
      binding="netEventRelayBinding"
      bindingConfiguration="default"
      behaviorConfiguration="sharedSecret"
      address=
        "sb://WCFBook.servicebus.windows.net/Subscriber"/>
  </client>

  <services>
    <service name="SubscriberService">
      <endpoint name="RelayEndpoint"
        contract="ISubscriber"
        binding="netEventRelayBinding"
        bindingConfiguration="default"
        behaviorConfiguration="sharedSecret"
        address=
          "sb://WCFBook.servicebus.windows.net/Subscriber"/>
    </service>
  </services>

  <behaviors>
    <endpointBehaviors>
      <behavior name="sharedSecret">
        <transportClientEndpointBehavior
            credentialType="SharedSecret">
          <clientCredentials>
            <sharedSecret issuerName="[ISSUER]"
                issuerSecret="[SECRET]"/>
          </clientCredentials>
        </transportClientEndpointBehavior>
      </behavior>
    </endpointBehaviors>
  </behaviors>
</system.serviceModel>
```

Привязка NetTcpRelayBinding

NetTcpRelayBinding представляет собой привязку, которую Microsoft рекомендует применять всегда, когда это возможно, так как она обеспечивает семантику двунаправленного обмена сообщениями и тесно связана со стандартной привязкой NetTcpBinding (основное отличие заключается в том, что NetTcpRelayBinding создает открытый общедоступный адрес в службе ретрансляции).

По умолчанию эта привязка использует протокол SOAP 1.2 над TCP с применением бинарного кодирования для более эффективного использования сети.

Кроме того, она поддерживает два режима подключения (табл. 10.3), управляющих тем, как клиент и сервер обмениваются информацией через службу ретрансляции.

Таблица 10.3. Режимы подключения NetTcpRelayBinding

Режим	Описание
Relayed	Используется по умолчанию. Все подключения ретранслируются через службу ретрансляции. После установки соединения служба ретрансляции работает в качестве прокси перенаправления сокетов, ретранслируя двунаправленный поток байтов
Hybrid	Первоначальное сообщение передается через службу ретрансляции, когда клиент и служба пытаются установить прямое соединение сокетов между собой. Все "переговоры" об установке соединения обеспечиваются службой ретрансляции. После того как прямое соединение может быть установлено, существующее соединение будет автоматически обновлено до прямого без потери сообщений или данных. Если прямое соединение не может быть установлено, данные будут продолжать проходить через службу ретрансляции как обычно

Когда включен режим по умолчанию Relayed, все входящие сообщения, поступающие на один из интерфейсных узлов, генерируют управляющие сообщения, которые маршрутизируются и направляются локальной службе WCF с инструкциями для создания обратного синхронизирующего соединения с интерфейсным узлом клиента. Когда это выполнено, для ретранслируемых сообщений TCP устанавливается прямая пересылка между сокетами.

С другой стороны, режим Hybrid поручает службе ретрансляции сделать все возможное, чтобы установить прямую связь между приложениями клиента и службы, так, чтобы данным больше не нужно было проходить через службу ретрансляции. Название "гибридный" указывает, как именно работает привязка в данном случае. Работа начинается с ретрансляции информации, в то же время выполняются попытки перейти на прямую связь. Если прямое соединение может быть установлено, переход к нему осуществится без потери данных. В противном случае будет продолжать использоваться служба ретрансляции. В этом режиме используется специальный алгоритм прогнозирования портов, основанный на пробной информации, получаемой от клиента и службы. Служба ретрансляции исследует эту информацию и пытается выяснить, какие порты в соответствующих устройствах NAT открыты. Затем она может предоставить сделанные выводы клиенту и службе, чтобы они могли попытаться установить прямую связь друг с другом. Если информация верна, соединение будет успешно установлено; в противном случае служба ретрансляции будет повторять попытки до тех пор, пока не придет к выводу о невозможности установки непосредственного соединения, и станет поддерживать ретрансляцию для данного соединения без дальнейших попыток обеспечить прямую связь.

Приведенная ниже конфигурация демонстрирует, как настроить службу WCF с единственной конечной точкой NetTcpRelayBinding в режиме Hybrid.

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="clientCredentials">
        <transportClientEndpointBehavior
            credentialType="SharedSecret">
          <clientCredentials>
            <sharedSecret issuerName="ISSUER_NAME"
                          issuerSecret="ISSUER_SECRET" />
          </clientCredentials>
        </transportClientEndpointBehavior>
      </behavior>
    </endpointBehaviors>
  </behaviors>

  <bindings>
    <netTcpRelayBinding>
      <binding name="default" connectionMode="Hybrid">
        <security mode="None" />
      </binding>
    </netTcpRelayBinding>
  </bindings>

  <services>
    <service name="HelloWorldService">
      <endpoint
        address="sb://WCFBook.servicebus.windows.net/HelloWorld"
        behaviorConfiguration="clientCredentials"
        binding="netTcpRelayBinding"
        bindingConfiguration="default"
        name="RelayEndpoint"
        contract="IHelloWorld" />
    </service>
  </services>
</system.serviceModel>
```

Режим `Hybrid` обеспечивает также свойство канала клиента WCF, заключающееся в том, что клиентское приложение может подписаться на события изменений состояния соединения во время выполнения. Это свойство реализовано в типе `IHybrid-ConnectionStatus` и предоставляет событие `ConnectionStateChanged`, для которого клиентское приложение может зарегистрировать обработчик. Приведенный ниже фрагмент кода показывает, как клиентское приложение может подписаться на это событие.

```
ChannelFactory <IHelloWorld> channelFactory =
        new ChannelFactory <IHelloWorld> ("RelayEndpoint");
IHelloWorld channel = channelFactory.CreateChannel();
channel.Open();

IHybridConnectionStatus hybridConnectionStatus =
        channel.GetProperty <IHybridConnectionStatus> ();

hybridConnectionStatus.ConnectionStateChanged += ( o,e ) =>
{
    Console.WriteLine("Connection Upgraded!!!!");
};
```

Можете также рассмотреть вопрос о применении `WS-ReliableMessaging` с `NetTcpRelayBinding`, так как это гарантирует восстановление соединения от вашего имени при возникновении проблем с прямым соединением сокетов. Это создает впечатление надежности прямых соединений.

Привязка `HttpRelayBinding`

Все привязки ретрансляции, рассмотренные к этому моменту, требуют от клиента использования WCF для работы с локальными службами Service Bus.

Service Bus также поставляется с несколькими привязками HTTP — `WebHttpRelayBinding`, `BasicHttpRelayBinding` и `WS2007HttpRelayBinding` — для поддержки сценариев, в которых необходима интеграция клиентов, не являющихся клиентами WCF. Эти привязки обеспечивают более широкий охват и бо́льшую способность к взаимодействию, потому что могут поддерживать любого клиента, который в состоянии использовать стандартные протоколы, поддерживаемые каждой из этих привязок.

Хотя привязка `WS2007HttpRelayBinding` обеспечивает более богатые функциональные возможности посредством поддержки протоколов `WS-*`, она требует, чтобы клиент хорошо понимал работу этих протоколов при взаимодействии клиента и службы. С другой стороны, `WebHttpRelayBinding` и `BasicHttpRelayBinding` основаны на простых протоколах, HTTP/REST и SOAP соответственно, так что в этом случае легче обеспечить совместимость.

Независимо от того, какая из привязок Http ретрансляции используется, все они работают примерно одинаково. Локальная служба WCF сначала подключается к службе ретрансляции с использованием либо соединения TCP, либо соединения HTTP в зависимости от выбранного значения `SystemConnectivityMode`.

После этого клиенты начинают отправлять сообщения в конечную точку Http, предоставляемую службой ретрансляции. Все, что надо для работы, — это простая клиентская библиотека Http или SOAP.

Служба ретрансляции может маршрутизировать любые сообщения SOAP 1.1, SOAP 1.2 или простые сообщения Http. Вам только нужно указать, какой стиль сообщений или протоколы `WS-*` вы можете использовать с помощью одной из привязок ретрансляции Http. Давайте реализуем простую RESTful службу Http, предоставляемую Service Bus.

Локальная служба WCF предоставляет простой контракт службы RESTful с помощью модели веб-программирования WCF 3.5 (листинг 10.9).

Листинг 10.9. Реализация службы RESTful


```
[ServiceContract()]
public interface ICustomerManagement
{
    [OperationContract]
    [WebGet]
    string GetCustomerName(int customerId);
}

public class CustomerManagement : ICustomerManagement
{
    public string GetCustomerName(int customerId)
    {
        if (customerId > 5)
        {
```

```
            return "Foo";
        }
        return "Bar";
    }
}
```

Реализация `GetCustomerName` просто возвращает `"Foo"` или `"Bar"` в зависимости от полученного аргумента `customerId`.

Размещающее приложение сконфигурировано с единственной конечной точкой `WebHttpRelayBinding` (листинг 10.10). Конечная точка службы сконфигурирована с учетными данными `"SharedSecret"`, а привязка `WebHttpRelayBinding` сконфигурирована таким образом, чтобы не требовать аутентификации от входящих клиентов. Это означает, что клиенты не должны проходить аутентификацию Access Control для использования вашей конечной точки.

Листинг 10.10. Конфигурация службы


```
<system.serviceModel>
  <bindings>
    <webHttpRelayBinding>
      <binding name="default">
        <security relayClientAuthenticationType="None"/>
      </binding>
    </webHttpRelayBinding>
  </bindings>
  <services>
    <service name="CustomerManagement"
             behaviorConfiguration="default">
      <endpoint name="RelayEndpoint"
        contract="ICustomerManagement"
        binding="webHttpRelayBinding"
        bindingConfiguration="default"
        behaviorConfiguration=
            "sharedSecretClientCredentials"
        address=
            "http://WCFBook.servicebus.windows.net/Customer"/>
    </service>
  </services>
  <behaviors>
    <endpointBehaviors>
      <behavior name="sharedSecretClientCredentials">
        <transportClientEndpointBehavior
            credentialType="SharedSecret">
          <clientCredentials>
            <sharedSecret issuerName="ISSUER_NAME"
                          issuerSecret="ISSUER_SECRET"/>
          </clientCredentials>
        </transportClientEndpointBehavior>
      </behavior>
    </endpointBehaviors>
    <serviceBehaviors>
      <behavior name="default">
        <serviceDebug httpHelpPageEnabled="false"
                      httpsHelpPageEnabled="false"/>
```

```
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Когда запущено и работает приложение службы WCF, любой может воспользоваться браузером и обратиться по адресу `http://WCFBook.servicebus.windows.net/CustomerManagement/GetCustomerName` с идентификатором `CustomerID` в строке запроса (например, `GetCustomerName?CustomerID=3`), чтобы получить имя клиента.

Работа со службой Access Control (ACS)

Служба управления доступом (Access Control Service — ACS) является службой "облака", которая обеспечивает централизованную проверку аутентификации и авторизации для вашего приложения и локальных служб посредством модели безопасности на базе утверждений. Вам не нужно предоставлять пользовательский код или иметь базу данных пользователей для ее использования, так как ACS берет все это на себя. Вы можете разрешить доступ как корпоративных, так и простых клиентов с помощью нескольких строк кода. Чем больше пользователей со временем начинают применять ACS, тем для большего количества пользователей ваших приложений можно будет проверить их подлинность или обеспечить для них доступ, ничего не меняя в коде.

ACS работает следующим образом. Пользователь должен получить маркер безопасности (SAML или Simple Web Token) от ACS, для того, чтобы войти в вашу службу или приложение. Этот маркер подписан ACS и содержит набор утверждений о личности пользователя. ACS не выдаст маркер, пока пользователь не пройдет успешно аутентификацию, предоставляя поддерживаемые учетные данные клиента (см. табл. 10.1).

В текущей версии ACS выдает только маркеры Simple Web Tokens, и основной причиной этого является то, что Microsoft решила сделать эту первую версию целевой только для служб RESTful. В то же время этот вид служб стал весьма популярным как в веб, так и среди корпоративных разработчиков. Они считают, что на рынке технологий контроля доступа появилась не заполненная ниша. Разработчики сталкиваются с отсутствием общих моделей для идентификации и управления доступом, совместимых с REST и достаточно простых в работе. Основная цель ACS — заполнить указанные пустоты.

Сегодня в ACS имеются три основные части: служба маркеров, выдающая маркеры безопасности, портал администрирования, а также API администрирования RESTful для настройки способа генерации создаваемых маркеров. SDK также включает утилиту командной строки Access Control Management Tool (`acm.exe`), которую можно использовать для выполнения операций по управлению (`CREATE`, `GET`, `UPDATE` и `DELETE`) объектами AppFabric Access Control (областями видимости, издателями, политиками маркеров и правилами). Объекты Access Control обсуждаются более подробно в следующих разделах.

Пространства имен служб

Пространства имен служб в контексте ACS — не более чем контейнер для объектов ACS. Они работают практически так же, как и в Service Bus, но в данном случае объекты специфичны для ACS. Пространство имен службы имеет собственную STS и управление конечными точками.

Некоторые из объектов в пространстве имен службы Access Control — правила. Правило определяет, какие утверждения будут добавлены в маркеры, выпущенные STS ACS. Каждое пространство имен службы может иметь совершенно иной набор правил. Правила играют очень важную роль, потому что большинству приложений, вероятно, понадобятся различные типы утверждений. Одному приложению может потребоваться проверить полное имя пользователя, так что можно ожидать требования, чтобы в маркер, выдаваемый ACS, было включено это полное имя. Другому приложению, возможно, потребуется другая личная информация. Очевидно, что каждое приложение имеет свой набор утверждений, а следовательно, и свой собственный набор правил, чтобы ACS могла генерировать эти утверждения. Эта идея настолько важна, что сообщения-запросы для получения маркера от ACS должны включать поле `AppliesTo`, которое представляет собой URI, идентифицирующий логическое назначение маркера. ACS использует его, позволяя назначать URI каждому из ваших приложений.

На рис. 10.6 показана структура пространства имен службы в ACS. Пространство имен службы `WCFBook` принадлежит учетной записи типа Windows Live ID `WCFBook@live.com` и содержит издателей, области видимости и правила.

Рис. 10.6. Структура пространства имен службы

Области видимости

Каждому приложению или службе в пределах пространства имен службы ACS должен быть назначен уникальный URI (вспомните уже рассматривавшееся ранее поле `AppliesTo`). Хотя технически это может быть любая строка, представляющая корректный URI, большинству людей более удобно использовать связанный с приложением URL, например `http://localhost/MyService`. Есть два очевидных преимущества в применении URL приложения в качестве `AppliesTo`: сразу становится очевидным, к какому приложению относится это имя, а кроме того, при таком подходе трудно случайно назначить один и тот же URI двум различным приложениям.

Каждое пространство имен службы ACS может управлять несколькими приложениями, так что каждое приложение должно иметь свою собственную область видимости с настройками и утверждениями, имеющими смысл для данного приложения.

Область видимости включает в себя набор правил, которые определяют, как именно ACS генерирует утверждения для данного приложения, а также включает другие важные параметры, такие как ключ, который ACS должна использовать для подписания маркеров безопасности, и их время жизни (как долго они будут действительны).

Время жизни маркера является очень важным с точки зрения масштабируемости и производительности. После издания маркера ACS он может быть использован

клиентом до истечения срока его действия многократно, что снижает количество запросов нового маркера клиентом от издателя, уменьшая тем самым время ожидания для клиентов и нагрузку издателя.

Как уже говорилось в главе 9, важную роль также играет подпись, так как позволяет службе убедиться в том, что источник маркера надежен и что маркер не был перехвачен и изменен.

Эти два параметра входят в ACS как часть политики работы с маркером. Эта политика может быть связана с несколькими областями видимости.

Издатели

Пространство имен службы поддерживает также одного или нескольких издателей. *Издатель* (issuer) в ACS представляет собой клиента, который будет запрашиваться для получения маркеров для работы со службой. Каждый издатель включает также набор мандатов, в том числе ключ. Можно использовать имя издателя и ключ просто для аутентификации издателя и запроса маркера, но можно решать и существенно более сложные задачи с использованием существующих маркеров.

Правила

Правила играют важную роль в службе AC, поскольку они управляют выдачей маркеров. Правило всегда связано с областью видимости и в основном отображает входные утверждения на выходные. Входные утверждения в общем случае создаются в процессе аутентификации, и действия правила состоят в преобразовании их в некоторые другие выходные утверждения, ожидаемые службой или приложением, получающим маркер.

В табл. 10.4 приведены все свойства, которые могут являться частью правила.

Таблица 10.4. Свойства правил

Свойство	Описание
`Name`	Дружественное имя для идентификации правила в пределах области видимости
`Type`	Указывает тип преобразования, которое правило применяет ко входным утверждениям. Два возможных значения —`Simple` и `Pass-through`. Преобразование `Simple` получает входное утверждение и создает совершенно иное выходное (иного типа и с иным значением). Преобразование `Pass-through` просто изменяет тип выходного утверждения
`Input Claim-Issuer`	Указывает издателя выходного утверждения. Это правило применимо только ко входным утверждениям, генерируемым данным издателем
`Input Claim-Type`	Строка, представляющая тип утверждения. Например, корректными значениями данного свойства являются `FullName` и `Email`
`Input Claim-Value`	Строка, представляющая значение утверждения. Например, для типа утверждения `Email` возможным значением может быть `"foo@mail.com"`
`Output Claim-Type`	Строка, представляющая тип выходного утверждения
`Output Claim-Value`	Строка, представляющая значение выходного утверждения

ACS будет выполнять все правила для данной области видимости, соответствующие свойствам `Issuer`, `Type` и `Value` входных утверждений. Результатом будет совершенно иной набор выходных правил.

Скажем, служба ожидает набор утверждений типа `"action"`, указывающий, какие действия потребителю позволено выполнять на стороне службы. Правило может

создать все эти утверждения, основываясь на способе аутентификации клиента в ACS. Например, в табл. 10.5 показано возможное правило для данной службы.

Таблица 10.5. Пример правила

Свойство	Значение
Name	ActionRule
Type	Simple
Input Claim – Issuer	MyClientApplication
Input Claim – Type	Issuer
Input Claim – Value	MyClientApplication
Output Claim – Type	Action
Output Claim – Value	DeleteCustomer

Когда клиентское приложение выполняет аутентификацию ACS, используя имя и пароль, связанный с издателем `MyClientApplication`, ACS будет автоматически генерировать входное утверждение `Issuer` с именем издателя в качестве значения. Это входное утверждение соответствует свойствам данного правила, поэтому к выдаваемому маркеру добавляется выходное утверждение `Action` со значением `DeleteCustomer`. Тип утверждения `Issuer` в данном случае представляет собой встроенный тип утверждения, который ACS генерирует, когда клиент проходит аутентификацию с именем и паролем издателя.

Интеграция вашей первой службы со службой Access Control

До сих пор мы теоретически рассматривали, что такое служба ACS и как ее можно использовать для обеспечения безопасности своих служб. Пришло время на более конкретном примере рассмотреть эту службу в действии.

В качестве первого примера используем пространство имен, которое создали при рассмотрении Service Bus (пространство имен службы `WCFBook`).

Контракт и реализация службы RESTful, которая будет представлена в Service Bus, представляет собой реализацию простейшей классической операции `SayHello`, которая возвращает фиксированной строку `"Hello"` (листинг 10.11).

Листинг 10.11. Реализация службы "Hello World"


```
[ServiceContract]
public interface ISampleService
{
    [OperationContract]
    [WebGet(UriTemplate = "sayHello")]
    string SayHello();
}

public class SampleService : ISampleService
{
    public string SayHello()
    {
        return "Hello";
    }
}
```

Поскольку для получения утверждений об аутентификации пользователей эта служба будет полагаться на ACS, вы будете использовать пользовательский диспетчер авторизации, который проверяет разрешения пользователя на основе соответствующих данных. Диспетчер авторизации будет вычислять все полученные утверждения, и клиент будет выполнять операцию только тогда, когда одно из этих утверждений соответствует одному из ожидаемых утверждений. Ожидаемые утверждения предварительно настраиваются размещающим приложением при инициализации диспетчера авторизации.

Таким образом, код консольного приложения, используемого для размещения службы, инициализирует следующее: принимающее приложение для службы WCF с привязкой `WebHttpBinding` для служб RESTful, рассмотренный ранее пользовательский диспетчер авторизации, а также криптографический ключ для проверки подписи маркера.

В листинге 10.12 показан код, необходимый для размещения службы и ожидающий некоторых утверждений от ACS. Как можно видеть, диспетчер авторизации ожидает выданный ACS маркер для службы "`http://localhost/SampleService`" (`AppliesTo`), который должен содержать по меньшей мере одно утверждение типа "`action`" со значением "`SayHello`".

Листинг 10.12. Реализация размещающего приложения службы

Доступно для загрузки на Wrox.com

```
class Service
{
  static void Main(string[] args)
  {
    string acsHostName = "accesscontrol.windows.net";

    string trustedAudience    =
            "http://localhost/SampleService";
    string requiredClaimType  = "action";
    string requiredClaimValue = "sayHello";

    Console.WriteLine(
        "Enter your service namespace, then press <ENTER> ");
    string serviceNamespace = Console.ReadLine();

    Console.WriteLine(
        "\nEnter your issuer key, then press <ENTER> ");
    string trustedTokenPolicyKey = Console.ReadLine();

    WebHttpBinding binding =
        new WebHttpBinding(WebHttpSecurityMode.None);

    Uri address = new Uri("http://localhost/SampleService");

    WebServiceHost host =
        new WebServiceHost(typeof(SampleService));
    host.AddServiceEndpoint(typeof(ISampleService),
                            binding, address);

    host.Authorization.ServiceAuthorizationManager =
        new ACSAuthorizationManager(
            acsHostName,
            serviceNamespace,
            trustedAudience,
            Convert.FromBase64String(trustedTokenPolicyKey),
            requiredClaimType,
```

```
            requiredClaimValue);

    host.Open();

    Console.WriteLine("The Sample Service is listening");
    Console.WriteLine("Press <ENTER> to exit");
    Console.ReadLine();
  }
}
```

Следующий шаг заключается в настройке правил и областей видимости в ACS для генерации утверждений, ожидаемых службой. В качестве части примера имеется пакетный файл `setup.cmd`, который автоматически делает все это для вас с помощью утилиты командной строки `ACS.exe`. Имеется также файл `cleanup.cmd`, который выполняет противоположные действия, удаляя все ранее созданные объекты.

Чтобы увидеть результаты выполнения этой команды в ACS, SDK включает в себя очень полезное приложение Windows `AcmBrowser`, которое использует API администрирования для наглядного представления обо всех объектах, сконфигурированных для определенного пространства имен. Образец можно найти в папке *`[Папка SDK]`*`\Samples\AccessControl\ExploringFeatures\Management`, созданной во время инсталляции SDK.

Чтобы получить доступ ко всей информации, хранимой в пространстве имен, при запуске этого приложения укажите имя этого пространства имен службы и управляющий ключ (доступный на странице пространства имен службы на веб-портале).

На рис. 10.7 показано, как `AcmBrowser` отображает все созданные объекты.

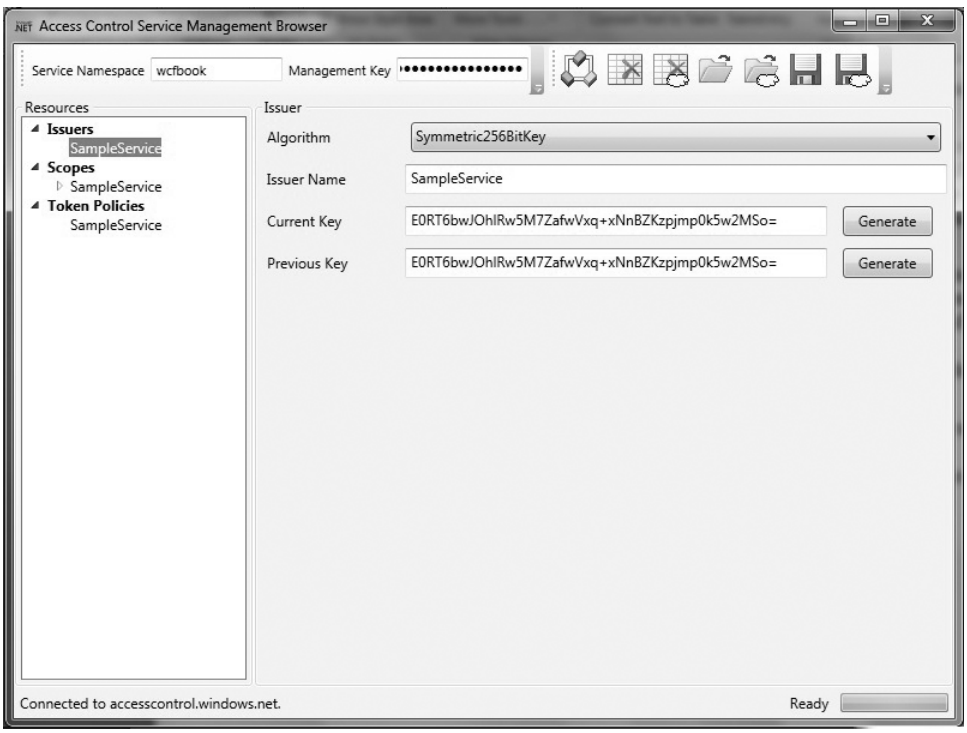


Рис. 10.7. Отображение всех созданных объектов

Пакетный файл генерирует нового издателя, SampleService, с ключом в формате Base64 для запрашиваемых маркеров, новую область видимости SampleService, которая относится к http://localhost/SampleService (URI для идентификации службы), и содержит простое правило для генерации выходного утверждения с разрешениями для клиента в виде утверждения типа "action".

Можете сгенерировать все эти объекты либо вручную, либо с помощью утилиты командной строки Acs.exe или AcmBrowser, но это означает, что для подготовки данного примера потребуется существенно больше работы на вашей стороне.

Диспетчер авторизации, обсуждавшийся ранее, реализует всю логику проверки маркера типа Simple Web Token, изданного ACS, и проверяет, содержит ли маркер требование, ожидаемое службой. В нашем примере сконфигурированная служба ожидает утверждения типа "action" и значения "SayHello".

Этот диспетчер авторизации реализован в основном в классе ACSAuthorization-Manager (листинг 10.13).

Листинг 10.13. Реализация диспетчера авторизации WCF


```
protected override bool
    CheckAccessCore(OperationContext operationContext)
{
  string authorizationHeader =
    WebOperationContext.Current.IncomingRequest.
        Headers[HttpRequestHeader.Authorization];
  if (string.IsNullOrEmpty(authorizationHeader))
  {
    return false;
  }

  if (!authorizationHeader.StartsWith("WRAP "))
  {
    return false;
  }

  string[] nameValuePair =
    authorizationHeader.Substring("WRAP ".Length).
        Split(new char[] {'='}, 2);

  if (nameValuePair.Length != 2 ||
      nameValuePair[0] != "access_token" ||
      !nameValuePair[1].StartsWith("\"") ||
      !nameValuePair[1].EndsWith("\""))
  {
    return false;
  }

  string token =
    nameValuePair[1].
    Substring(1, nameValuePair[1].Length - 2);

  if (!this.validator.Validate(token))
  {
    return false;
```

```
  }

  Dictionary <string, string> claims =
    this.validator.GetNameValues(token);

  string actionClaimValue;
  if (!claims.TryGetValue(this.requiredClaimType,
                          out actionClaimValue))
  {
    return false;
  }

  if (!actionClaimValue.Equals(this.requiredClaimValue))
  {
    return false;
  }

  return true;
}
```

Маркеры в основном извлекаются из заголовков авторизации HTTP, после чего с помощью вспомогательного класса `TokenValidator` (который также включен в рассматриваемый пример) выполняется несколько проверок, чтобы убедиться, что это корректный маркер типа Simple Web Token. В последней части реализации этого метода утверждения извлекаются из маркера и сравниваются с утверждением, ожидаемым службой.

После того как служба полностью реализована и настроена служба ACS, наступает время реализации клиентского приложения для работы со службой.

Реализация клиентского приложения включает в себя два основных этапа: согласование маркера с ACS и окончательное выполнение службы с ожидаемыми данными и согласованным маркером.

Чтобы получить от ACS маркер, клиентское приложение должно выполнить к службе запрос HTTP типа `POST`, включающий следующую информацию.

- Поле `wrap_name`, представляющее имя сконфигурированного издателя ACS.
- Поле `wrap_password`, представляющее ключ, назначенный издателю.
- Поле `wrap_scope`, представляющее значение поля `AppliesTo` для данной области видимости в ACS.

Эти три поля корректны только в случае аутентификации клиента с именем и паролем издателя. Необходимые поля варьируются согласно методу аутентификации, используемому при работе с ACS.

В листинге 10.14 показано согласование маркера с ACS.

Листинг 10.14. Согласование маркера с ACS


```
private static string GetTokenFromACS()
{
  WebClient client = new WebClient();
  client.BaseAddress =
    string.Format("https://{0}.{1}", serviceNamespace,
                  acsBaseAddress);
```

```
  NameValueCollection values = new NameValueCollection();
  values.Add("wrap_name", "SampleService");
  values.Add("wrap_password", issuerKey);
  values.Add("wrap_scope","http://localhost/SampleService");

  byte[] responseBytes =
    client.UploadValues("WRAPv0.9", "POST", values);

  string response = Encoding.UTF8.GetString(responseBytes);

  Console.WriteLine("\nreceived token from ACS: {0}\n",
                    response);

  return response
    .Split(' & ')
    .Single(value = > value.StartsWith("wrap_access_token=",
                StringComparison.OrdinalIgnoreCase))
    .Split('=')[1];
}
```

После получения маркера клиентское приложение может включить его в качестве части сообщения запроса к службе в виде заголовка авторизации HTTP (заголовок, ожидаемый диспетчером авторизации).

В листинге 10.15 показана работа со службой.

Листинг 10.15. Работа со службой


```
private static string SendMessageToService(string token)
{
  string message = null;

  WebHttpBinding binding =
    new WebHttpBinding(WebHttpSecurityMode.None);
  Uri address = new Uri(@"http://localhost/SampleService");

  WebChannelFactory < ISampleService > channelFactory = new
WebChannelFactory < ISampleService > (binding, address);

  ISampleService proxy = channelFactory.CreateChannel();

  using (new OperationContextScope(proxy as IContextChannel))
  {
    string authHeaderValue =
        string.Format("WRAP access_token=\"{0}\"",
                      HttpUtility.UrlDecode(token));

    WebOperationContext.Current.
        OutgoingRequest.Headers.Add("authorization",
                                    authHeaderValue);

    message = proxy.SayHello();
  }

  ((IClientChannel)proxy).Close();
```

```
  channelFactory.Close();

  return message;
}
```

Этот метод использует `WebChannelFactory` для работы со службой RESTful, поэтому заголовок авторизации может быть добавлен посредством `OperationContextScope`. Этот код может иметь иной вид, если вы используете другую библиотеку для работы с Http.

11

Создание примера работы с SOA

В ЭТОЙ ГЛАВЕ...

- Определение требований для примера
- Разработка полного решения
- Тестирование решения

Эта глава представляет собой интерактивное пошаговое руководство по созданию решения Visual Studio 2010 в архитектуре SOA с помощью WCF. Она начинается с определения требований для данного решения и шаг за шагом проводит вас по его разработке. В конце главы вы разработаете ряд служб, приложений для их размещения (хостов) и клиентских приложений, являющихся частью решения. Это полный пример, и вы можете протестировать его в действии.

Требования

Вам нужно создать службы для компании, которая занимается арендой автомобилей. Компании требуется служба для управления автопарком, службы регистрации клиентов и регистрации договоров аренды.

Эти три службы рассматриваются как внутренние и будут использоваться собственными приложениями.

Помимо этих служб, требуется также внешняя служба, доступная для партнеров компании, которая позволяет с помощью одного вызова добавить клиента и арендовать автомобиль.

Операции `CarManagementService`.

- `InsertNewCar`. Получает данные об автомобиле и вставляет их в базу данных.
- `RemoveCar`. Получает идентификатор автомобиля и удаляет соответствующую запись из базы данных.
- `UpdateMileage`. Получает данные об автомобиле и обновляет информацию о его пробеге в базе данных.
- `ListCars`. Возвращает все автомобили с их данными.
- `GetCarPicture`. Возвращает изображение автомобиля для определенного идентификатора.

Операции `CustomerService`.

- `RegisterCustomer`. Получает данные клиента и вставляет их в базу данных.

Операции `RentalService`.

- `RegisterCarRental`. Получает всю информацию для регистрации договора аренды.
- `RegisterCarRentalAsPaid`. Получает идентификатор договора аренды и регистрирует его в базе данных как оплаченный.
- `StartRental`. Вызывает метод для указания того факта, что автомобиль был арендован в данном месте.
- `StopCarRental`. Вызывает метод для указания того факта, что автомобиль возвращен в данном месте, где завершена его аренда.
- `GetRentalRegistration`. Возвращает всю информацию о договоре аренды.

Операции `ExternalService`.

- `SubmitRentalContract`. Получает всю информацию о новом клиенте и договоре аренды. Вносит в базу данных нового клиента и договор аренды.

Дополнительные требования

Дополнительные требования к `CarManagementService`.

- Наличие поля перечислимого типа для указания ручной или автоматической коробки передач.
- Количество типов автомобилей в компании. Помимо обычных автомобилей, имеются представительские и спортивные автомобили. Представительские автомобили включают все элементы данных обычного автомобиля и список элементов представительского класса. Спортивные автомобили, помимо элементов данных обычных автомобилей, включают мощность двигателя.

Дополнительные требования к `RentalService`.

- Аккуратная обработка ошибок. Данные должны быть проверены, и если введены некорректные данные, клиент должен получить понятное сообщение с описанием правила проверки.
- Код операции `RegisterCarRentalAsPaid` должен выполняться от имени учетной записи клиента, использующего данную службу.

Дополнительные требования к `ExternalService`.

- `SubmitRentalContract` последовательно вызывает операции двух других служб (`CustomerService` и `RentalService`) для регистрации клиента и договора аренды. Очевидно, что это должно осуществляться в пределах единой транзакции. Клиент не должен вноситься в базу данных без регистрации договора аренды.

Настройка решения

Работа начинается с создания пустого решения Visual Studio, в котором будут создаваться все необходимые проекты. Это упрощает тестирование всех приложений и позволяет содержать их в единой системе управления версиями (рис. 11.1).

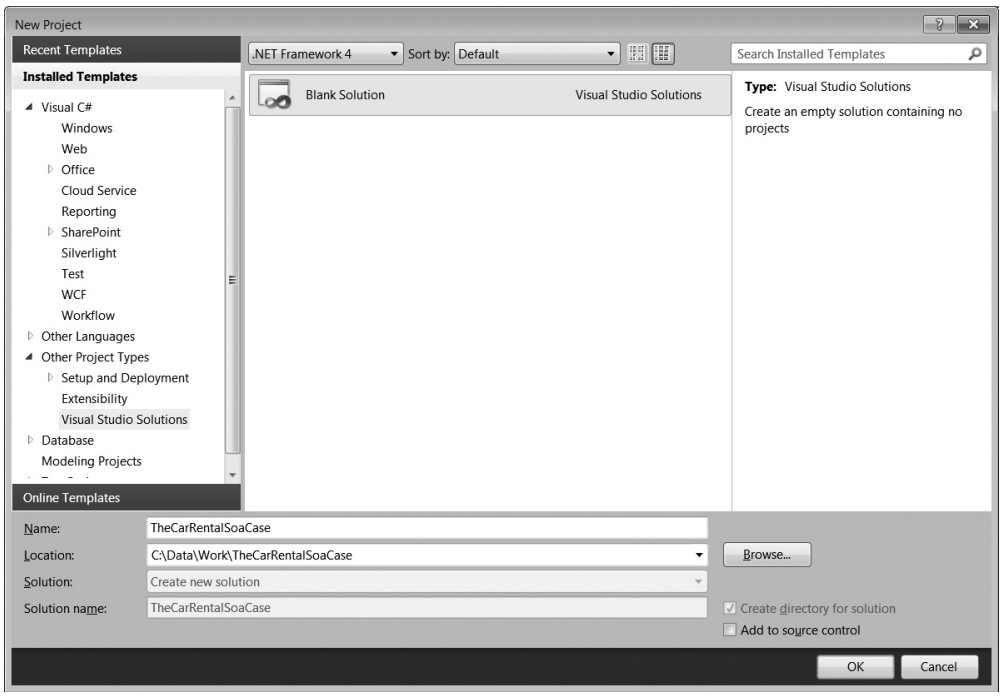


Рис. 11.1. Новое решение Visual Studio

Назовите решение `TheCarRentalSOACase` и разместите его в каталоге `C:\Data\Work\TheCarRentalSOACase`.

Создайте структуру папок решения для группирования проектов. Каждая папка будет содержать один или несколько проектов. Можете добавить папку решения щелчком правой кнопкой мыши на решении с последующим выбором пункта контекстного меню Add⇨New Solution Folder (рис. 11.2).

Требуются следующие папки решения.

- Клиенты (имя папки — `Clients`).
- Размещающие приложения (имя папки — `Hosts`).
- Интерфейсы (имя папки — `Interfaces`).
- Службы (имя папки — `Services`).

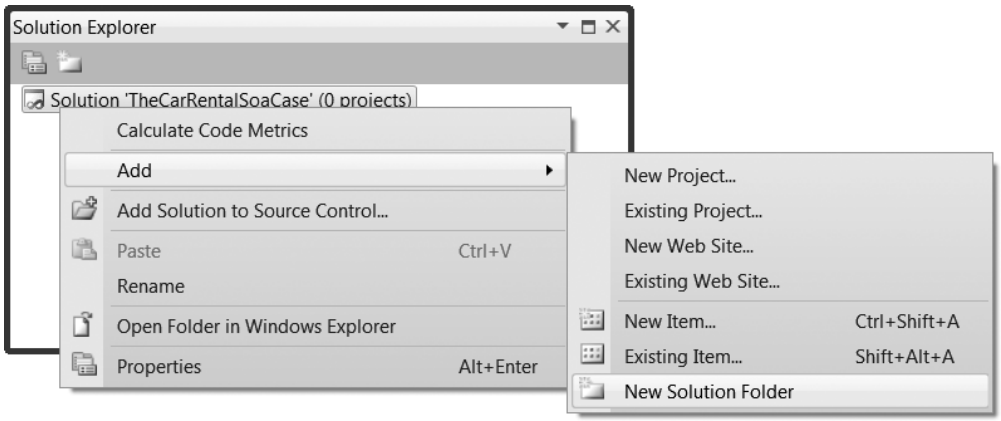


Рис. 11.2. Добавление новой папки

После добавления указанных четырех папок решение будет иметь вид, показанный на рис. 11.3.

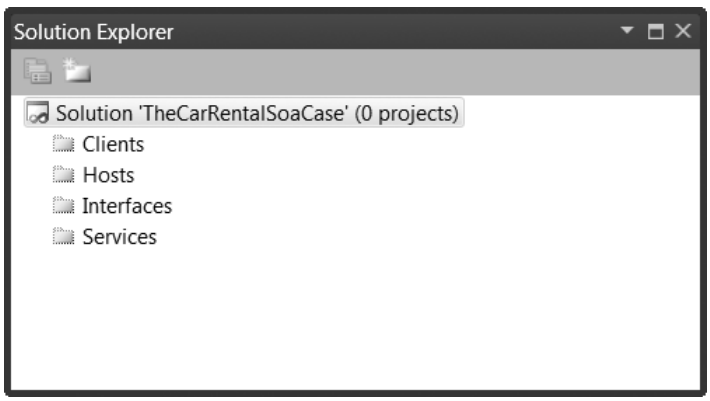


Рис. 11.3. Решение после добавления необходимых папок

Создание интерфейсов

Лучше всего в случае проекта SOA начинать с создания интерфейсов (и это достаточно распространенная практика). Начнем с создания библиотек, содержащих интерфейсы. Нам нужны четыре библиотеки:

- `CarManagementInterface`;
- `CustomerInterface`;
- `RentalInterface`;
- `ExternalInterface`.

Создать библиотеки можно путем добавления четырех проектов Class Libraries в папку `Interfaces` решения (рис. 11.4).

В диалоговом окне Add New Project из списка шаблонов выберите Class Library (рис. 11.5).

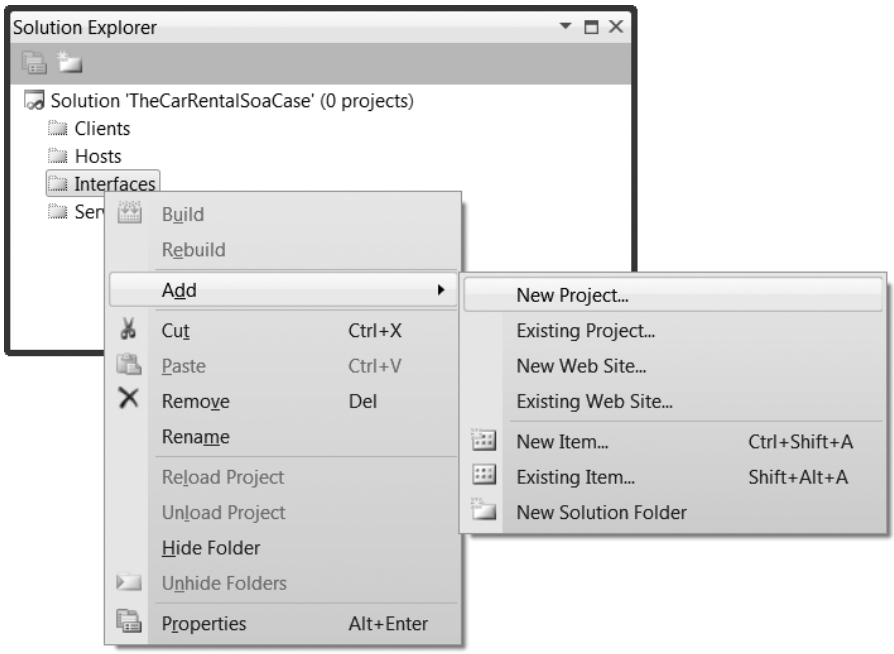


Рис. 11.4. Добавление четырех проектов в папку `Interfaces`

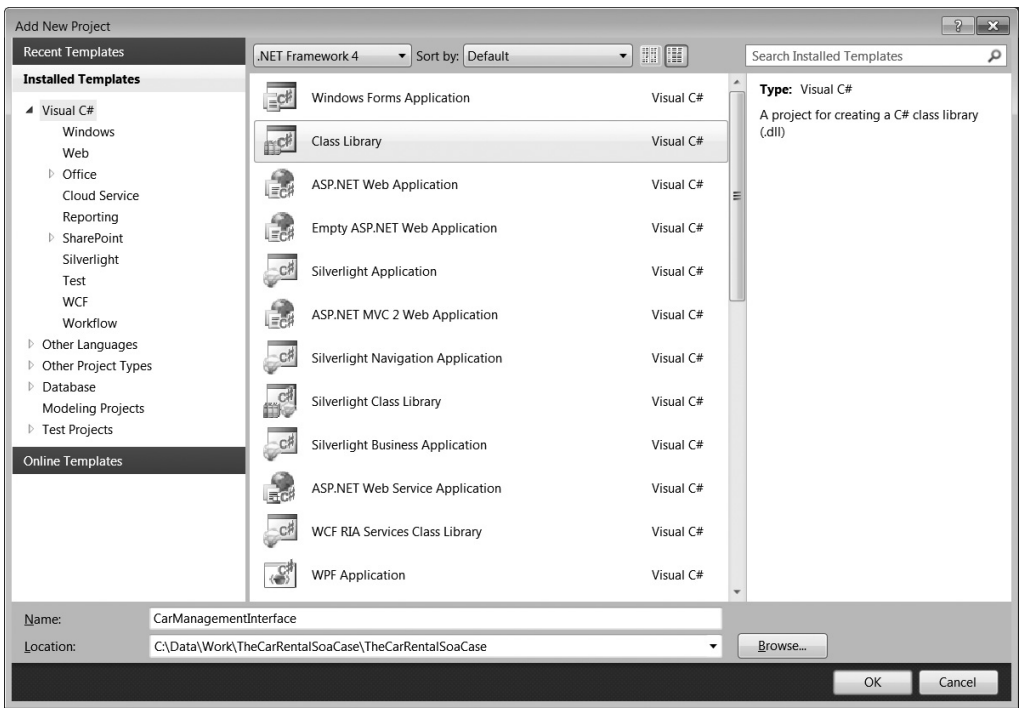


Рис. 11.5. Диалоговое окно Add New Project

Выполните это действие для каждого из четырех интерфейсов. Ваше решение должно выглядеть так, как показано на рис. 11.6.

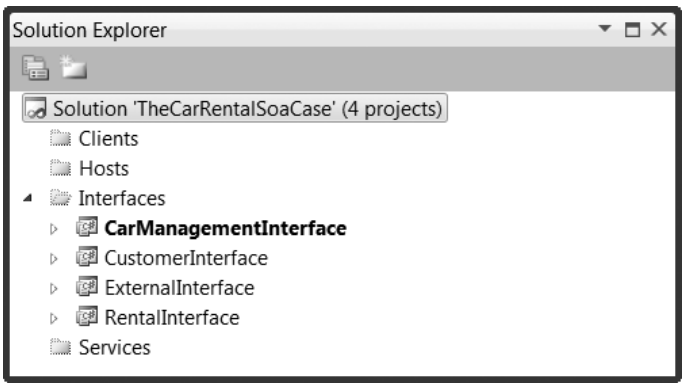


Рис. 11.6. Текущее состояние решения

Теперь необходимо добавить ссылки на библиотеки WCF `System.ServiceModel` и `System.Runtime.Serialization`. Они содержат атрибуты, необходимые для контрактов. Добавление ссылок выполняется щелчком правой кнопкой мыши на пункте References проекта в окне Solution Explorer (рис. 11.7).

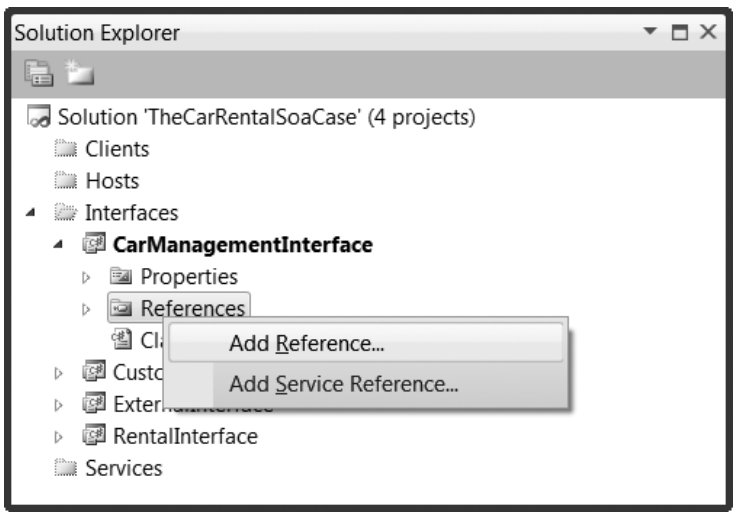


Рис. 11.7. Добавление ссылок

В открывшемся диалоговом окне Add Reference выберите `System.ServiceModel` и `System.Runtime.Serialization`, как показано на рис. 11.8, и щелкните на кнопке OK.

Повторите этот шаг для трех других проектов.

Поскольку в качестве шаблона выбрана библиотека классов Class Library, Visual Studio создает в проекте файл `Class1.cs`, содержащий пустой класс. Нам этот файл не нужен, так как мы создаем интерфейсы. Таким образом, можно просто удалить файл `Class1.cs` в каждом из четырех проектов (рис. 11.9).

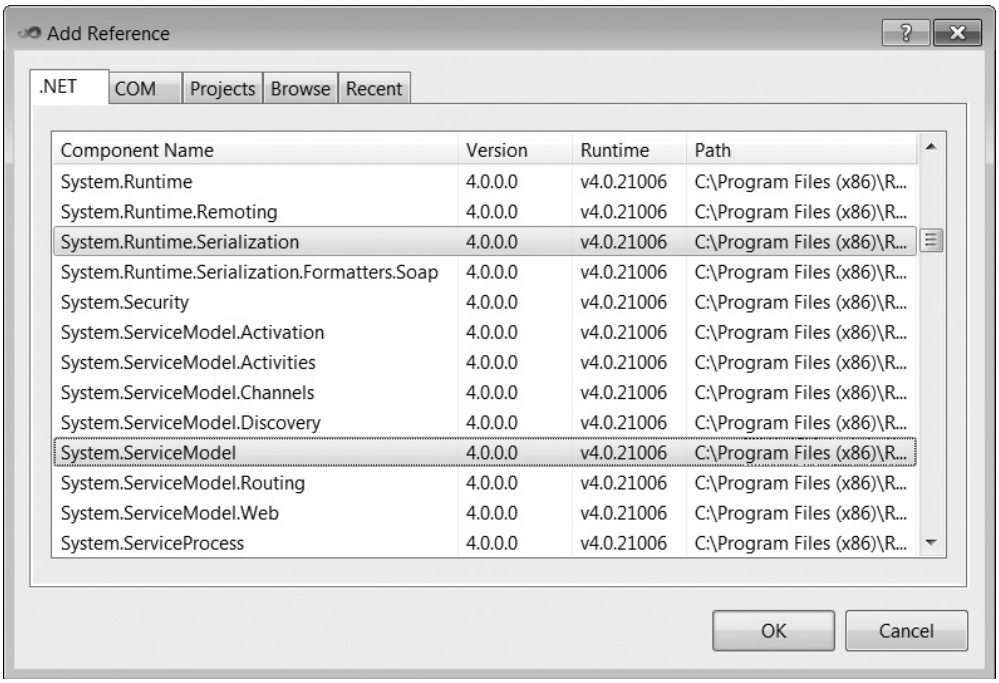


Рис. 11.8. Выбор библиотек WCF `System.ServiceModel` *и* `System.Runtime.Serialization`

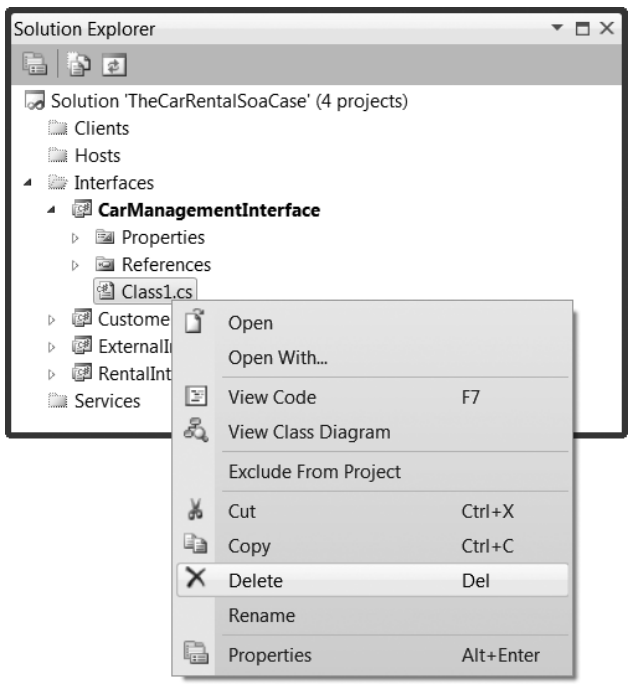


Рис. 11.9. Удаление файла `Class1.cs`

Далее добавьте к каждому проекту файл интерфейса (рис. 11.10). Это интерфейсы `ICarManagement`, `ICustomer`, `IRental` и `IExternalInterface`.

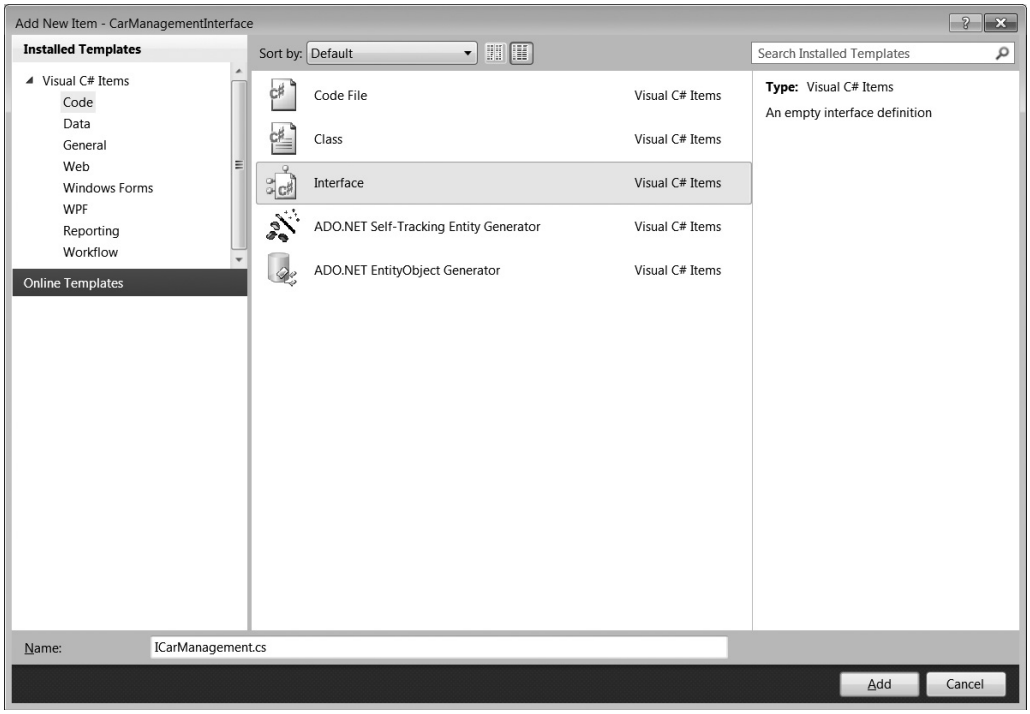


Рис. 11.10. Добавление файлов интерфейса в каждый проект

Откройте интерфейс и добавьте необходимые инструкции `using` для доступа к пространствам имен `System.ServiceModel` и `System.Runtime.Serialization`.

```
using System.ServiceModel;
using System.Runtime.Serialization;
```

Сделайте интерфейс открытым, добавляя в его объявление ключевое слово `public`.

```
public interface ICarManagement
```

Повторите эти действия для каждого проекта интерфейса.

Создание интерфейса `CarManagement`

Для выполнения требования о наличии поля перечислимого типа для коробки передач добавьте следующий исходный код.

```
[DataContract]
public enum TransmissionTypeEnum
{
    [EnumMember]
    Manual,
    [EnumMember]
    Automatic
}
```

Это — перечисление `public enum` с именем `TransmissionTypeEnum`. Перечисление должно иметь атрибут `DataContract`, а каждый его элемент — атрибут `EnumMember`.

Теперь добавьте код класса `Car`.

```
[DataContract]
public class Car
{
    [DataMember]
    public string BrandName { get; set; }
    [DataMember]
    public string TypeName { get; set; }
    [DataMember]
    public TransmissionTypeEnum Transmission { get; set; }
    [DataMember]
    public int NumberOfDoors { get; set; }
    [DataMember]
    public int MaxNumberOfPersons { get; set; }
    [DataMember]
    public int LitersOfLuggage { get; set; }
}
```

Этот класс помечен атрибутом `DataContract`, а каждое его свойство имеет атрибут `DataMember`.

Теперь можно завершить создание интерфейса, содержащего сигнатуры методов службы.

```
[ServiceContract]
interface ICarManagement
{
    [OperationContract]
    int InsertNewCar(Car car);
    [OperationContract]
    bool RemoveCar(Car car);
    [OperationContract]
    void UpdateMilage(Car car);
    [OperationContract]
    List < Car > ListCars();
    [OperationContract]
    byte[] GetCarPicture(string carID);
}
```

Исправьте все допущенные при вводе ошибки (если таковые имеются) и завершите работу, скомпилировав библиотеку классов.

Создание интерфейса `Customer`

Добавьте код к интерфейсу `ICustomer`. Это код класса `Customer` и сигнатуры операций в `CustomerService`.

```
[ServiceContract]
public interface ICustomer
{
    [OperationContract]
    int RegisterCustomer(Customer customer);
}

[DataContract]
```

```
public class Customer
{
    [DataMember]
    public string CustomerName { get; set; }
    [DataMember]
    public string CustomerFirstName { get; set; }
    [DataMember]
    public string CustomerMiddleLetter { get; set; }
    [DataMember]
    public DateTime CustomerBirthDate { get; set; }
}
```

Создание интерфейса `Rental`

Добавьте код к интерфейсу `IRental`. Это код контракта `RentalRegistration` и сигнатуры операций в `RentalService`.


```
[ServiceContract]
public interface IRental
{
    [OperationContract]
    string RegisterCarRental(RentalRegistration rentalRegistration);

    [OperationContract]
    void RegisterCarRentalAsPaid(string rentalID);

    [OperationContract]
    void StartCarRental(string rentalID);

    [OperationContract]
    void StopCarRental(string rentalID);

    [OperationContract]
    RentalRegistration GetRentalRegistration(string rentalID);
}

[DataContract]
public class RentalRegistration
{
    [DataMember]
    public int CustomerID { get; set; }
    [DataMember]
    public string CarID { get; set; }
    [DataMember]
    public int PickUpLocation { get; set; }
    [DataMember]
    public int DropOffLocation { get; set; }
    [DataMember]
    public DateTime PickUpDateTime { get; set; }
    [DataMember]
    public DateTime DropOffDateTime { get; set; }
    [DataMember]
    public PaymentStatusEnum PaymentStatus { get; set; }
    [DataMember]
    public string Comments { get; set; }
}
```

```
[DataContract]
public enum PaymentStatusEnum
{
    [EnumMember(Value = "PUV")]
    PaidUpFrontByVoucher,
    [EnumMember(Value = "PUC")]
    PaidUpFrontByCreditCard,
    [EnumMember(Value = "TPP")]
    ToBePaidAtPickUp,
    [EnumMember(Value = "INV")]
    ToBePaidByInvoice
}

[DataContract]
public enum IncludedInsurance
{
    [EnumMember]
    LiabilityInsurance = 1,
    [EnumMember]
    FireInsurance = 2,
    [EnumMember]
    TheftProtection = 4,
    [EnumMember]
    AllRiskInsurance = 1 + 2 + 4
}
```

Файл CreatingaSOACase.zip

Создание внешнего интерфейса

Проект External Interface повторно использует контракты `Customer` и `Rental`, а потому требует обращения к обеим библиотекам. Откройте диалоговое окно Add Reference, перейдите на вкладку Projects и выберите элементы `CustomerInterface` и `RentalInterface` (рис. 11.11).

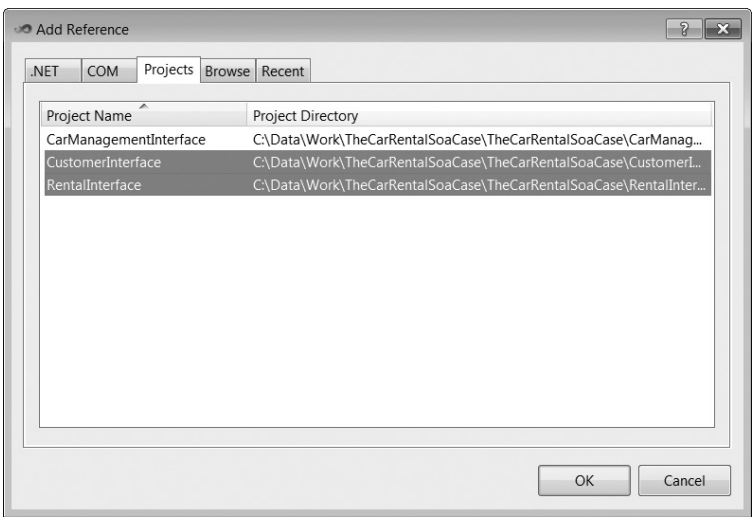


Рис. 11.11. Выбор элементов `CustomerInterface` и `RentalInterface`

Помимо добавления инструкций `using` для двух системных библиотек, добавьте инструкции `using` для `RentalInterface` и `CustomerInterface`.

```
using RentalInterface;
using CustomerInterface;
```

Ниже приведен код для `ExternalInterface`.

```
[ServiceContract]
public interface IExternalInterface
{
    [OperationContract]
    void SubmitRentalContract(RentalContract rentalContract);
}


[DataContract]
public class RentalContract
{
    [DataMember]
    public string Company { get; set; }
    [DataMember]
    public string CompanyReferenceID { get; set; }
    [DataMember]
    public RentalRegistration RentalRegistration { get; set; }
    [DataMember]
    public Customer Customer { get; set; }
}
```

Класс `ExternalInterface` имеет один метод, получающий параметр. Он структурирован контрактом, который имеет свойства `RentalRegistration` и `Customer`.

Создание служб

Добавим в решение четыре проекта библиотек классов, по одному для каждой службы. Эти проекты ссылаются на проекты интерфейсов и содержат реализацию их логики. Назовем их `CarManagementService`, `CustomerService`, `ExternalInterfaceFacade` и `RentalService`. Решение должно выглядеть так, как показано на рис. 11.12.

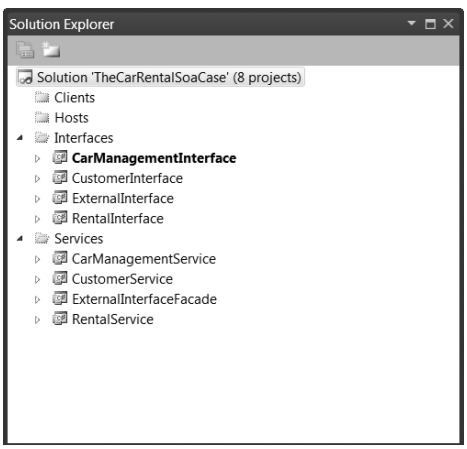


Рис. 11.12. Текущий вид решения

В каждый проект добавьте сборки `System.ServiceModel` и `System.Runtime.Serialization`.

Переименуйте файл `Class1.cs` в каждом проекте. Имена файлов после переименования — `CarManagementImplementation.cs`, `CustomerServiceImplementation.cs`, `RentalServiceImplementation.cs` и `ExternalInterfaceFacadeImplementation.cs`.

Когда Visual Studio запросит, не хотите ли вы переименовать элемент кода `Class1`, щелкните на кнопке Yes (рис. 11.13).

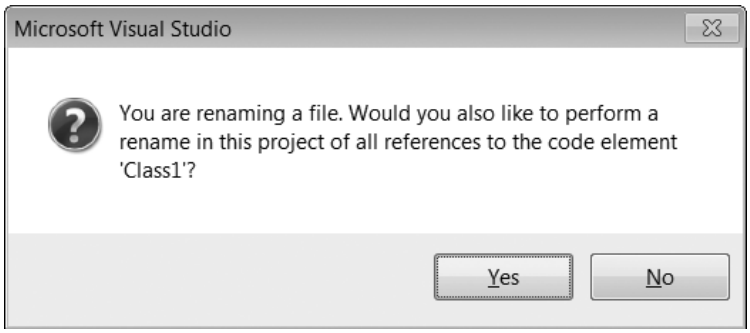


Рис. 11.13. Запрос на переименование элемента кода `Class1`

Результат показан на рис. 11.14.

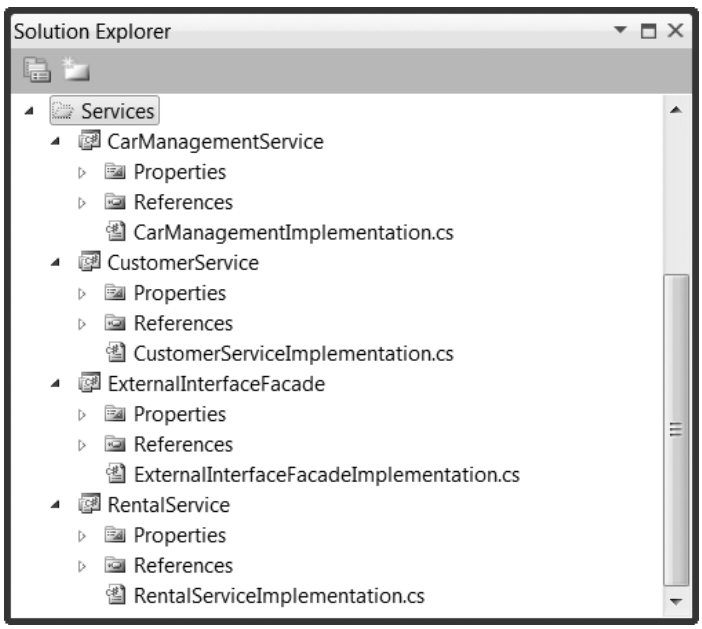


Рис. 11.14. Результат переименования

Добавьте в каждый файл необходимые инструкции `using`.

```
using System.Runtime.Serialization;
using System.ServiceModel;
```

В каждом проекте добавьте ссылку на соответствующий проект, содержащий интерфейс. Затем добавьте в каждый класс соответствующую инструкцию `using` для доступа к интерфейсу.

```
using CarManagementInterface;
```

Реализуйте интерфейс класса, добавляя двоеточие и имя интерфейса после объявления класса.

```
public class CarManagementImplementation : ICarManagement
{
}
```

Все методы интерфейса можно достаточно быстро реализовать щелчком правой кнопкой мыши на только что введенном имени интерфейса в редакторе исходного текста и выбором в контекстном меню пункта Implement Interface (рис. 11.15).

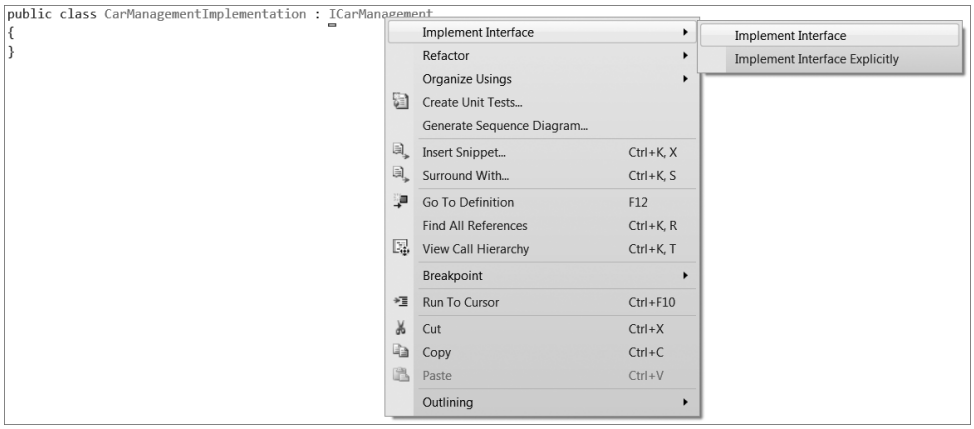


Рис. 11.15. Выбор в контекстном меню пункта Implement Interface

В случае интерфейса `CarManagement` результат должен иметь следующий вид.

```
public int InsertNewCar(Car car)
{
    throw new NotImplementedException();
}
public bool RemoveCar(Car car)
{
    throw new NotImplementedException();
}
public void UpdateMilage(Car car)
{
    throw new NotImplementedException();
}
public List < Car > ListCars()
{
    throw new NotImplementedException();
}
public byte[] GetCarPicture(string carID)
{
    throw new NotImplementedException();
}
```

Повторите эти действия для каждого из трех остальных интерфейсов, убедившись в том, что для каждой библиотеки реализации используется соответствующий интерфейс, а затем выполните сборку решения.

Создание хоста

Приступим к созданию консольного приложения для размещения службы — оно более удобно для отладки и тестирования. Позже вы сможете создать для размещения служб собственную службу Windows.

Добавьте в соответствующую папку решения приложение `Console` (рис. 11.16).

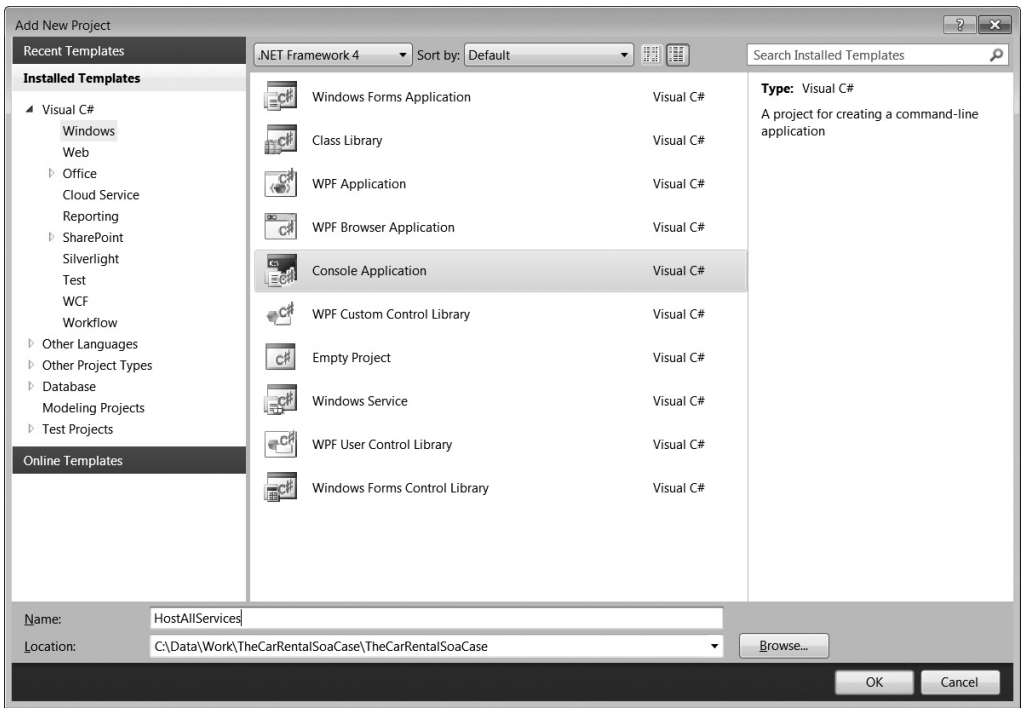


Рис. 11.16. Добавление в решение консольного приложения

Добавьте необходимые библиотеки WCF, а также все библиотеки интерфейсов и реализаций. Добавьте сборки `System.ServiceModel` и `System.Runtime.Serialization`, а затем все четыре проекта интерфейсов вместе с проектами реализаций (рис. 11.17).

К коду `program.cs` добавьте все необходимые инструкции `using` — две системные и четыре для проектов реализации.

```
using System.ServiceModel;
using System.Runtime.Serialization;
using CarManagementService;
using RentalService;
using CustomerService;
using ExternalInterfaceFacade;
```

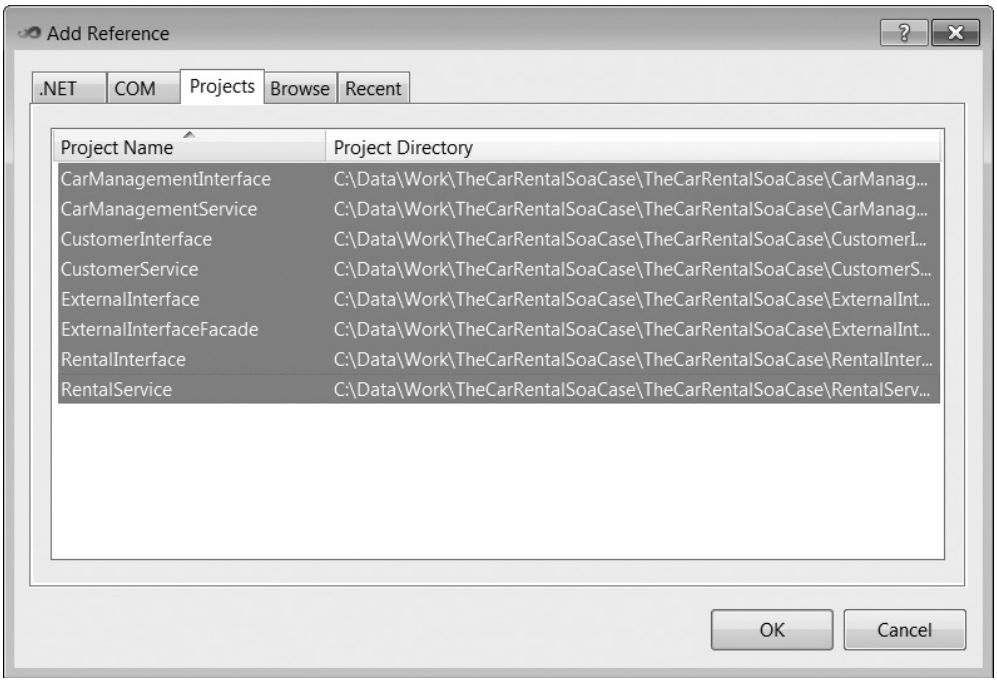


Рис. 11.17. Добавление четырех проектов интерфейсов вместе с проектами реализаций

Теперь добавьте в метод `main` код обработчика исключений вместе с инструкциями, выводящими на консоль состояние службы. Последняя строка содержит операцию `Console.ReadKey()`. Это гарантирует работу хоста после открытия служб, так что клиенты смогут обращаться к ним.

```
Console.WriteLine("ServiceHost");
try
{
    // Открываем размещающее приложение
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
Console.WriteLine("Started");
Console.ReadKey();
```

Нам нужны четыре хоста для служб. Объявите в классе программы ссылочную переменную для каждого из хостов.

```
static ServiceHost CarManagementServiceHost;
static ServiceHost CustomerServiceHost;
static ServiceHost RentalServiceHost;
static ServiceHost ExternalServiceHost;
```

Запишите код для инстанцирования и откройте каждый хост в блоке `try-catch` основного метода.

```
CarManagementServiceHost =
    new ServiceHost(typeof(CarManagementService.
                           CarManagementImplementation));
CarManagementServiceHost.Open();

CustomerServiceHost =
    new ServiceHost(typeof(CustomerService.
                           CustomerServiceImplementation));
CustomerServiceHost.Open();

RentalServiceHost =
    new ServiceHost(typeof(RentalService.
                           RentalServiceImplementation));
RentalServiceHost.Open();

ExternalServiceHost =
    new ServiceHost(typeof(ExternalInterfaceFacade.
                     ExternalInterfaceFacadeImplementation));
ExternalServiceHost.Open();
```

В диалоговом окне Add New Item добавьте в проект конфигурационный файл приложения. Выберите Application Configuration File и используйте имя по умолчанию — `App.config` (рис. 11.18).

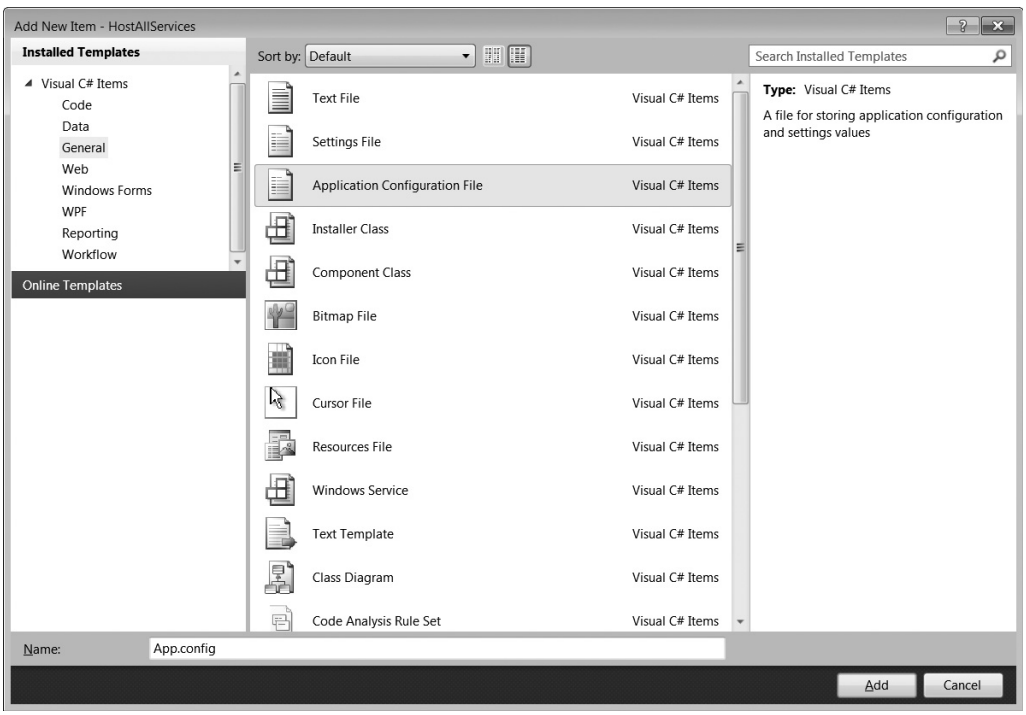


Рис. 11.18. Добавление в проект конфигурационного файла приложения

Теперь необходимо собрать все решение. Очень важно, чтобы хост был собран корректно. Процесс построения копирует сборки интерфейсов и реализаций в каталог

bin проекта. Эти бинарные файлы необходимы, поскольку WCF Configuration Editor при их конфигурировании должен иметь к ним доступ.

Щелкните правой кнопкой мыши на файле App.config и выберите Edit WCF Configuration (рис. 11.19-11.20).

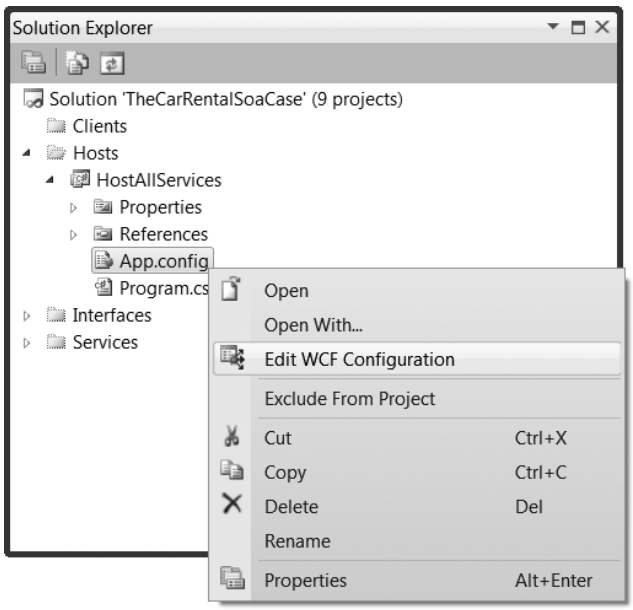


Рис. 11.19. Запуск редактора конфигурации

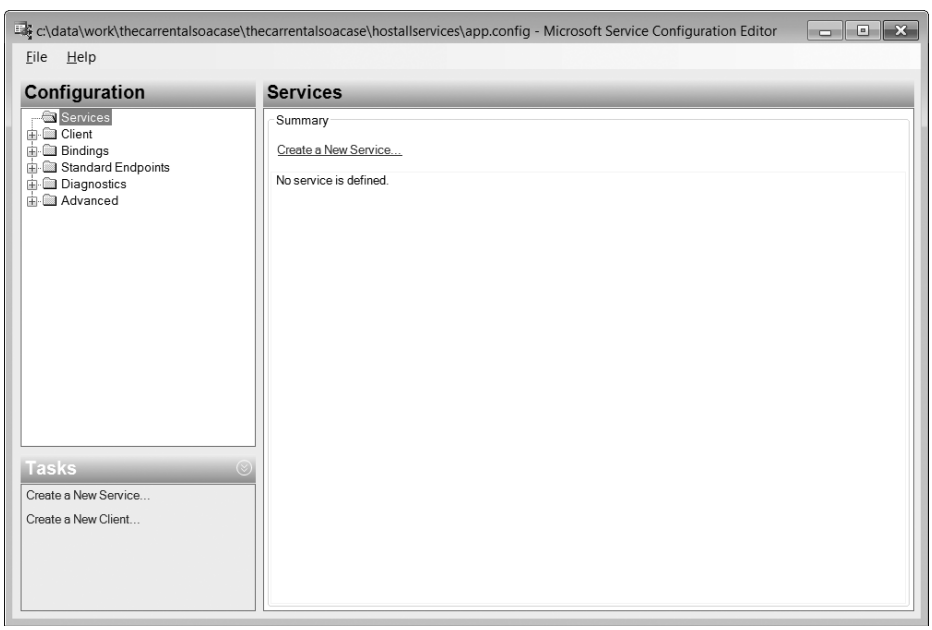


Рис. 11.20. Редактор конфигурации

В WCF Configuration Editor щелкните на Create a New Service (рис. 11.21).

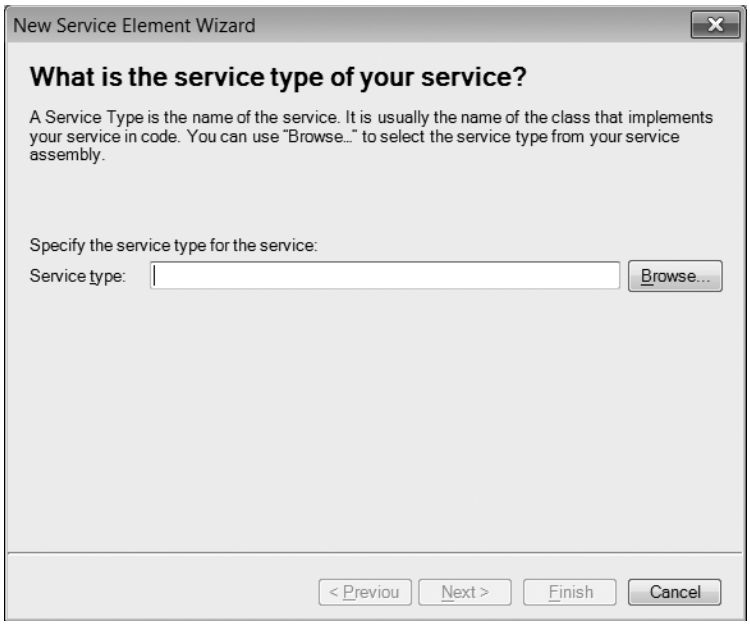


Рис. 11.21. Мастер создания нового элемента службы

Щелкните на кнопке Browse, чтобы открыть диалоговое окно Type Browser. Перейдите в каталог `bin`, а затем в каталог `debug`. В диалоговом окне выберите сборку `CarManagamentService.dll` и щелкните на кнопке Open (рис. 11.22).

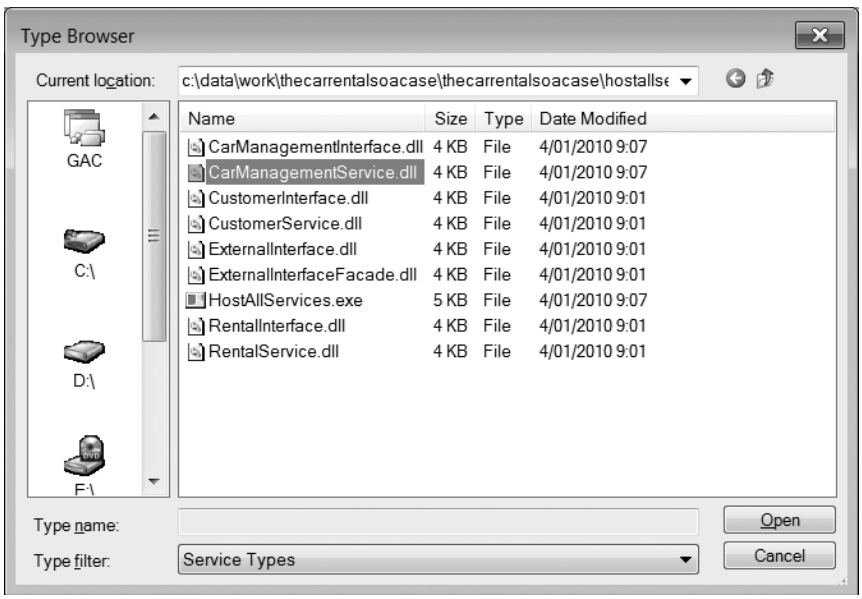


Рис. 11.22. Выбор сборки `CarManagamentService.dll`

При этом вы увидите класс реализации в этой сборке. Щелкните на `CarManagamentService.CarManagementImplementation`, а затем на кнопке Open (рис. 11.23).

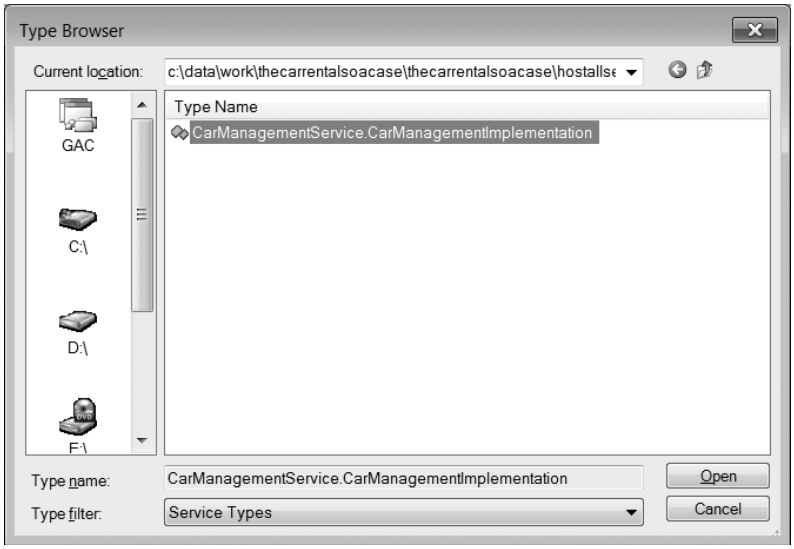


Рис. 11.23. Выбор класса `CarManagamentService.CarManagementImplementation`

Когда мастер Add Service предложит имя реализации в диалоговом окне, щелкните на кнопке Next.

Далее мастер примет решение об имени контракта. Не изменяйте его. Щелкните на кнопке Next.

Теперь мастер спросит, какой режим подключения использует служба. Оставьте выбранным HTTP (рис. 11.24) и щелкните на кнопке Next.

Рис. 11.24. Выбор режима подключения

После этого мастер запросит метод взаимодействия, как показано на рис. 11.25. Выберите Advanced Web Services interoperability, оставьте включенной настройку Simplex Communication и щелкните на кнопке Next.

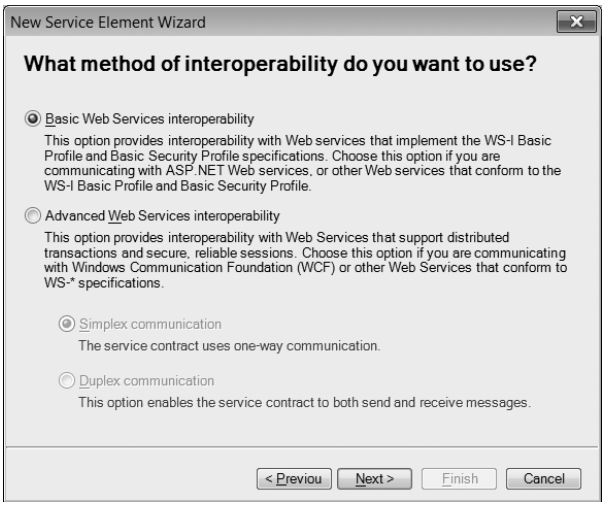


Рис. 11.25. Выбор метода взаимодействия

Далее мастер запросит адрес вашей конечной точки: `http://localhost:9876/CarManagementService` (рис. 11.26).

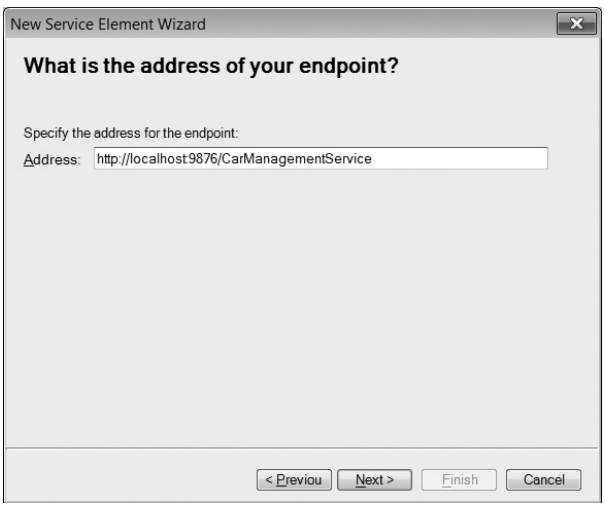


Рис. 11.26. Запрос адреса конечной точки

Щелкните на кнопках Next и Finish. Сохраните конфигурацию (File⇨Save) и выйдите из WCF Configuration Editor.

Если файл `App.config` был открыт в редакторе, Visual Studio обнаружит, что в него были внесены изменения, и выведет соответствующее диалоговое окно, показанное на рис. 11.27. Щелкните на кнопке Yes.

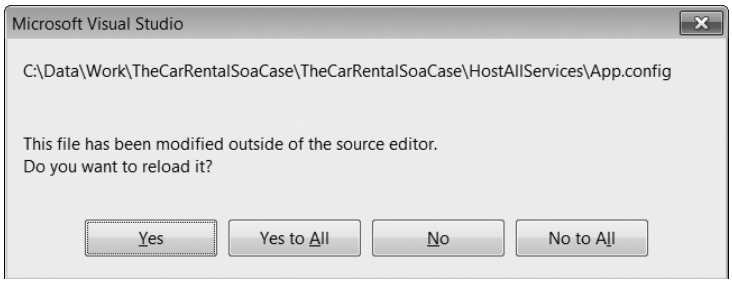


Рис. 11.27. Запрос среды разработки Visual Studio

Проверьте созданную конфигурацию. Мастер настраивает в качестве привязки к конечной точке `ws2007HttpBinding`. Измените ее на `wsHttpBinding`. Будьте внимательны — значение чувствительно к регистру! Результат должен иметь следующий вид.

```
<system.serviceModel>
  <services>
    <service
      name="CarManagementService.CarManagementImplementation">
      <endpoint
        address="http://localhost:9876/CarManagementService"
        binding="wsHttpBinding"
        bindingConfiguration=""
        contract="CarManagementInterface.ICarManagement" />
    </service>
  </services>
</system.serviceModel>
```

Вместо конфигурирования трех оставшихся служб по одной с помощью WCF Configuration Editor можно просто скопировать и вставить дескриптор `service`, а также заменить необходимые значения вручную. Используйте для остальных служб следующие адреса.

- `http://localhost:9876/CustomerService`
- `http://localhost:9876/RentalService`
- `http://localhost:9876/ExternalInterfaceService`

Убедитесь, что корректно изменили имя и контракт. Результат должен иметь следующий вид.


```
<system.serviceModel>
  <services>
    <service
      name="CarManagementService.CarManagementImplementation">
      <endpoint
        address="http://localhost:9876/CarManagementService"
        binding="wsHttpBinding" bindingConfiguration=""
        contract="CarManagementInterface.ICarManagement"/>
    </service>
    <service
      name="CustomerService.CustomerServiceImplementation">
      <endpoint
        address="http://localhost:9876/CustomerService"
```

```
        binding="wsHttpBinding" bindingConfiguration=""
        contract="CustomerInterface.ICustomer"/>
    </service>
    <service
      name="RentalService.RentalServiceImplementation">
      <endpoint
        address="http://localhost:9876/RentalService"
        binding="wsHttpBinding" bindingConfiguration=""
        contract="RentalInterface.IRental"/>
    </service>
    <service
      name="ExternalInterfaceFacade.ExternalInterfaceFacadeImplementation">
      <endpoint
        address="http://localhost:9876/ExternalInterfaceService"
        binding="wsHttpBinding" bindingConfiguration=""
        contract="ExternalInterface.IExternalInterface"/>
    </service>
  </services>
</system.serviceModel>
```

Файл CreatingaSOACase.zip

Установите `HostAllServices` в качестве стартового проекта. Для этого щелкните на проекте `HostAllServices` правой кнопкой мыши и выберите в контекстном меню Set as StartUp Project (рис. 11.28).

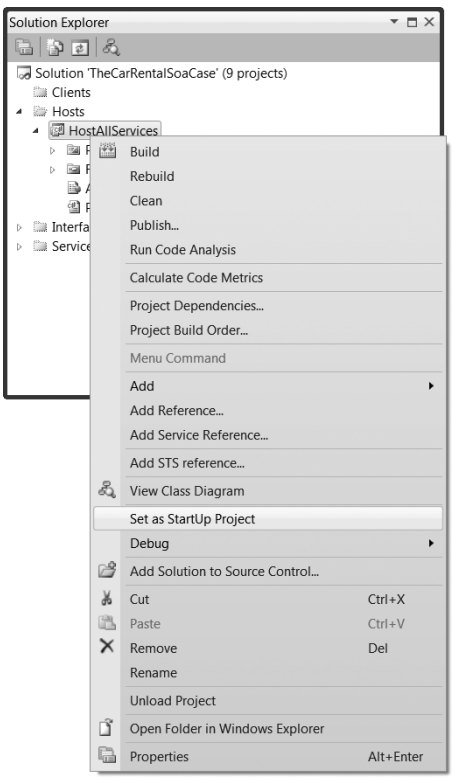


Рис. 11.28. Установка проекта `HostAllServices` в качестве стартового

Теперь можно запустить приложение, которое предоставляет созданные службы (рис. 11.29).

Рис. 11.29. Запуск хоста

Создание базы данных

Откройте Server Explorer в Visual Studio, создайте новую базу данных SQL Server Database (рис. 11.30) и назовите ее `RentalCarCaseDB`.

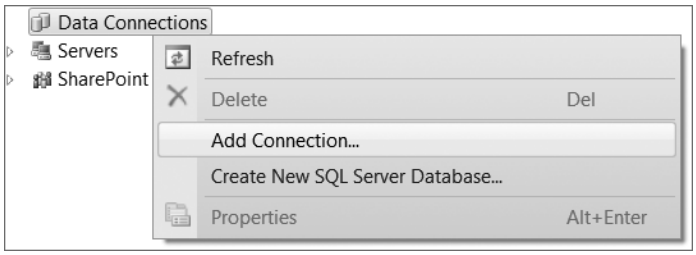


Рис. 11.30. Создание базы данных

Добавьте в базу данных две таблицы. Это можно сделать, выполнив приведенную ниже инструкцию SQL или добавив таблицу вручную.

```
CREATE DATABASE [CarRentalCaseDB]
GO
USE [CarRentalCaseDB]
GO
CREATE TABLE [dbo].[Rental]
(
    [RentalID] [int] IDENTITY(1,1) NOT NULL,
    [CustomerID] [int] NULL,
    [CarID] [nvarchar](50) NULL,
    [PickUpLocation] [int] NULL,
    [DropOffLocation] [int] NULL,
```

```
    [PickUpDateTime] [datetime] NULL,
    [DropOffDateTime] [datetime] NULL,
    [PaymentStatus] [char](3) NULL,
    [Comments] [nvarchar](1000) NULL,
    CONSTRAINT [PK_Rental] PRIMARY KEY CLUSTERED
        ([RentalID] ASC))
GO
CREATE TABLE [dbo].[Customer](
    [CustomerID] [int] IDENTITY(1,1) NOT NULL,
    [CustomerName] [nvarchar](50) NULL,
    [CustomerFirstName] [nvarchar](50) NULL,
    CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
        ([CustomerID] ASC))
GO
```

Реализация службы

Чтобы заставить службы работать, добавьте исходный код в их реализации.

Поскольку эта книга посвящена WCF, а не аренде автомобилей, здесь не будет представлена полная бизнес-логика служб. Вместо этого мы напишем временный код, чтобы операции возвращали некоторые осмысленные ответы клиентам. Для реализации `InsertCustomer` и `RegisterCarRental` воспользуемся встроенными в язык запросами (LINQ) для вставки данных в базу данных.

Создание доступа к базе данных для `CustomerService` и `RentalService`

Добавьте LINQ к файлам классов `CustomerService` и `RentalService` (рис. 11.31). В данном случае и `CustomerService`, и `RentalService` обращаются к одной и той же базе данных. В реальной ситуации это, вероятно, были бы две разные базы данных, но для нашего примера использование одной базы данных удобней. Добавьте в `CustomerService` LINQ to SQL Classes и назовите его `DataClassesCustomer.dbml` (рис. 11.32).

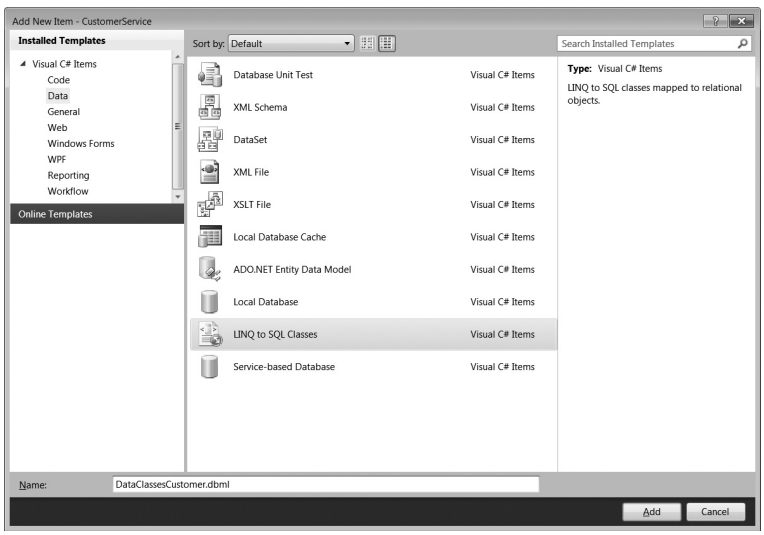


Рис. 11.31. Добавление класса LINQ to SQL

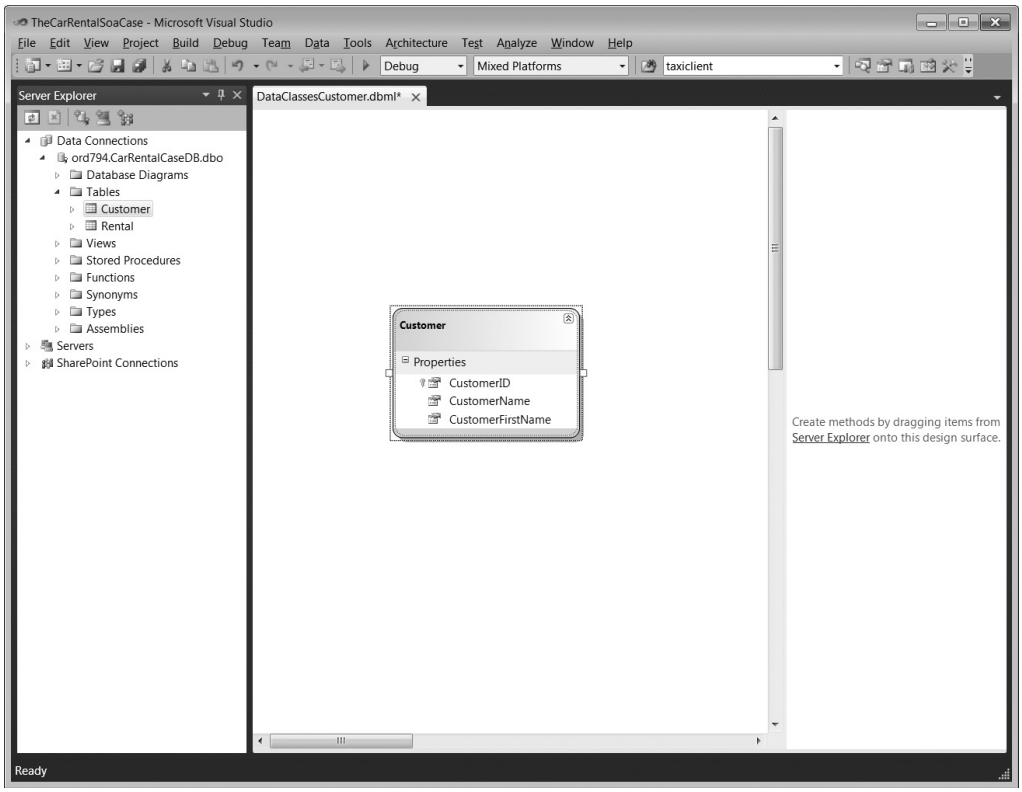


Рис. 11.32. База данных `DataClassesCustomer.dbml`

Перетащите таблицу `Customer` в проектировщик.

Теперь это же надо выполнить для `RentalService`. Создайте элемент LINQ to SQL model с именем `DataClassesRental.dbml` и перетащите в него таблицу `Rental`.

А сейчас можно заняться добавлением кода реализации в `CustomerService`. Откройте файл `CustomerServiceImplementation.cs`. В коде метода `RegisterCustomer` удалите инструкцию `throw` и замените ее следующим кодом.

```
using (DataClassesCustomerDataContext ctx =
            new DataClassesCustomerDataContext())
{
    Customer customerToInsert;
    customerToInsert = new Customer();
    customerToInsert.CustomerName = customer.CustomerName;
    customerToInsert.CustomerFirstName =
                        customer.CustomerFirstName;
    ctx.Customers.InsertOnSubmit(customerToInsert);
    ctx.SubmitChanges();
    return customerToInsert.CustomerID;
}
```

Этот код открывает LINQ to SQL DataContext, создает нового клиента и вносит его данные в базу данных.

Добавьте аналогичный код для внесения в базу данных информации о договоре аренды в реализацию RentalService. Откройте файл RentalServiceImplementation.cs и замените инструкцию throw следующим кодом.

```
Console.WriteLine("RegisterCarRental");
using (DataClassesRentalDataContext ctx =
            new DataClassesRentalDataContext())
{
    Rental rentalToInsert;
    rentalToInsert = new Rental();
    rentalToInsert.CustomerID = rentalRegistration.CustomerID;
    rentalToInsert.CarID = rentalRegistration.CarID;
    rentalToInsert.Comments = rentalRegistration.Comments;
    ctx.Rentals.InsertOnSubmit(rentalToInsert);
    ctx.SubmitChanges();
}
return "OK";
```

Создание службы CarManagement

Здесь мы добавим код в CarManagementService. Начнем с добавления инструкции using для пространства имен system.IO. Это пространство имен необходимо для работы с файлами.

```
using System.IO;
```

Реализуем операции с помощью приведенного ниже кода. Убедитесь, что в операции GetCarPicture вы используете путь к существующему файлу.

```
public int InsertNewCar(Car car)
{
    Console.WriteLine("InsertNewCar " + car.BrandName +
                      " " + car.TypeName);
    return 1;
}

public bool RemoveCar(Car car)
{
    Console.WriteLine("RemoveCar " + car.BrandName + " " +
                      car.TypeName);
    return true;
}

public void UpdateMilage(Car car)
{
    Console.WriteLine("UpdateMilage " + car.BrandName +
                      " " + car.TypeName);
}

public List < Car > ListCars()
{
    Console.WriteLine("ListCars");
    List < Car > listCars;
    listCars = new List < Car > ();
```

```
    listCars.Add(new Car {
        BrandName = "XXX",
        Transmission = TransmissionTypeEnum.Automatic,
        TypeName = "YYY" });
    listCars.Add(new Car {
        BrandName = "XXX",
        Transmission = TransmissionTypeEnum.Automatic,
        TypeName = "YYY" });
    return listCars;
}

public byte[] GetCarPicture(string carID)
{
    Console.WriteLine("GetCarPicture");

    byte[] buff;

    string pathToPicture;
    pathToPicture =
        @"C:\Data\WCFBook\Code\SOACase\Pics\CarExample.jpg";

    FileStream fileStream =
        new FileStream(pathToPicture,
                       FileMode.Open,FileAccess.Read);
    BinaryReader binaryReader = new BinaryReader(fileStream);

    buff = binaryReader.ReadBytes((int)fileStream.Length);
    return buff;
}
```

В этой реализации операции `InsertCar`, `RemoveCar` и `UpdateMileage` с базой данных не взаимодействуют — они просто выводят строки на консоль. Операция `ListCars` создает и возвращает жестко "зашитый" в код список автомобилей. Операция `GetCarPicture` считывает изображение из файла и возвращает его в виде массива байтов. Преобразование этого массива назад в изображение — задача клиента.

Работа с метаданными

Теперь можно разместить службу, добавляя конфигурацию для предоставления метаданных. Предоставление метаданных позволяет Visual Studio загрузить файл WSDL для создания необходимых прокси. Предоставление метаданных службой обеспечивается путем соответствующей конфигурации.

Мы сделаем это только для службы `CarManagementService`. В этой главе вы ознакомитесь и с другими способами создания прокси для иных служб.

Откройте файл `App.config` приложения `HostAllServices` в WCF Configuration Editor. Найдите узел Service Behaviors и щелкните на ссылке New Service Behavior Con figuration (рис. 11.33).

Назовите вновь создаваемую конфигурацию `ExposeMetaDataBehavior` (рис. 11.34).

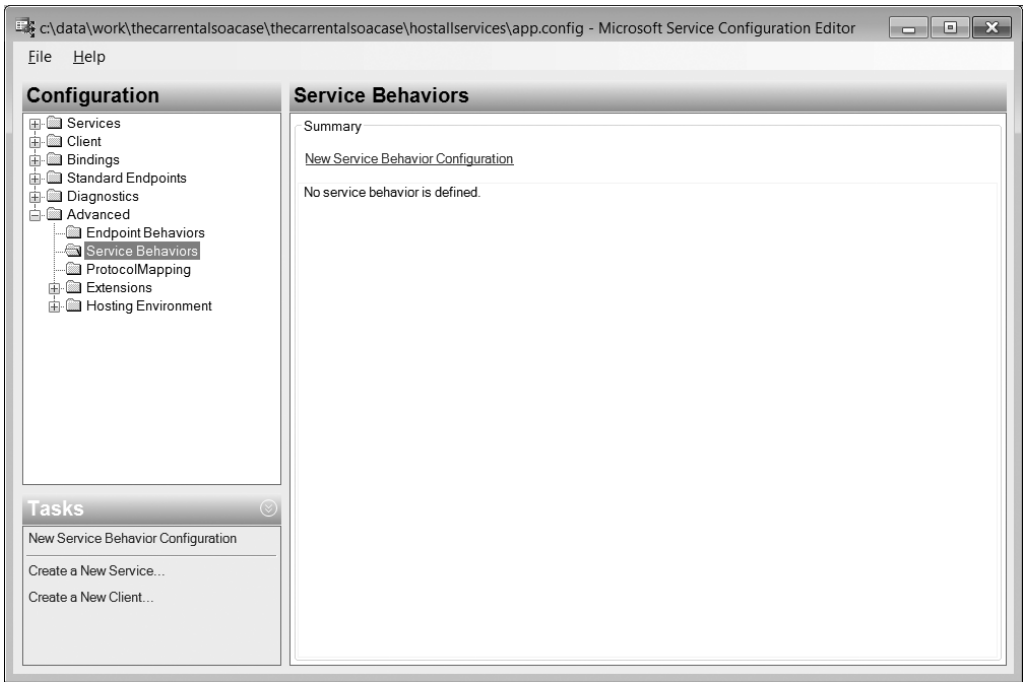


Рис. 11.33. Ссылка New Service Behavior Configuration

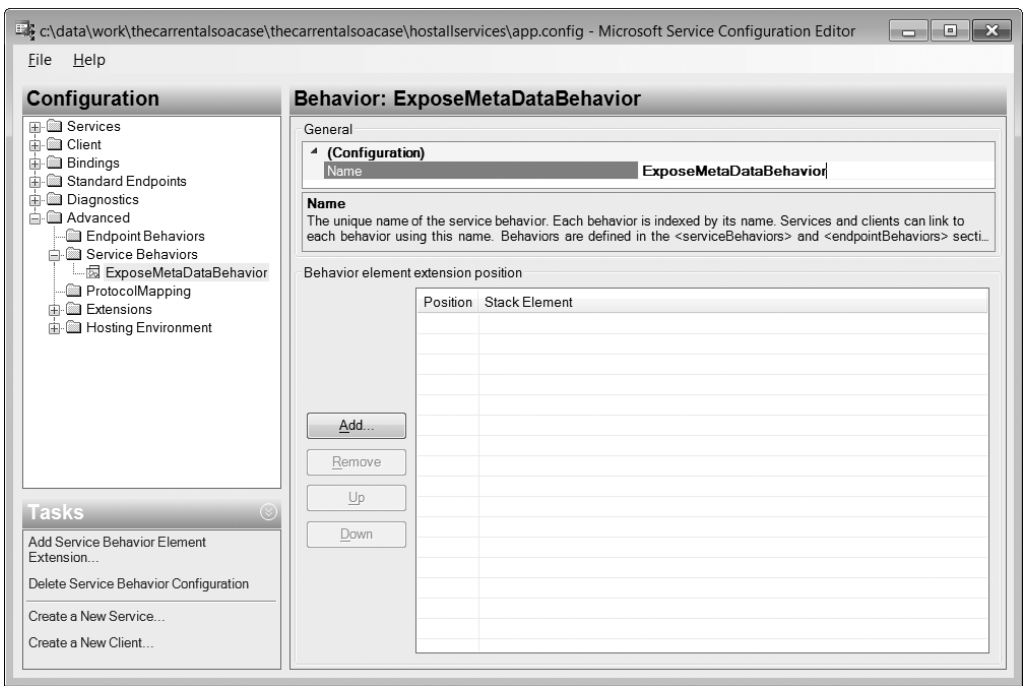


Рис. 11.34. Конфигурация `ExposeMetaDataBehavior`

Щелкните на кнопке Add, чтобы добавить элемент поведения. Откроется список элементов поведения, показанный на рис. 11.35.

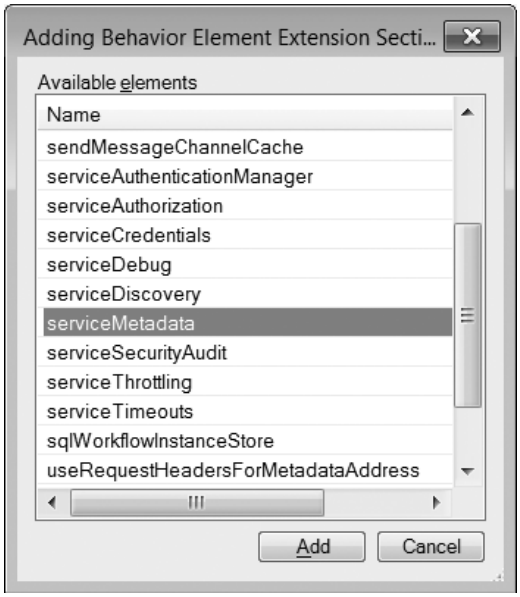


Рис. 11.35. Список элементов

Выберите элемент `serviceMetadata` и щелкните на кнопке Add. Обратите внимание на то, что элемент добавлен в узел `ExposeMetaDataBehavior` в древовидной структуре.

Щелкните на этом новом узле и отредактируйте атрибуты `serviceMetadata`. Введите `http://localhost:9876/CarManagement/MEX` в поле `HttpGetUrl` и установите для свойства `HttpGetEnabled` значение `True` (рис. 11.36).

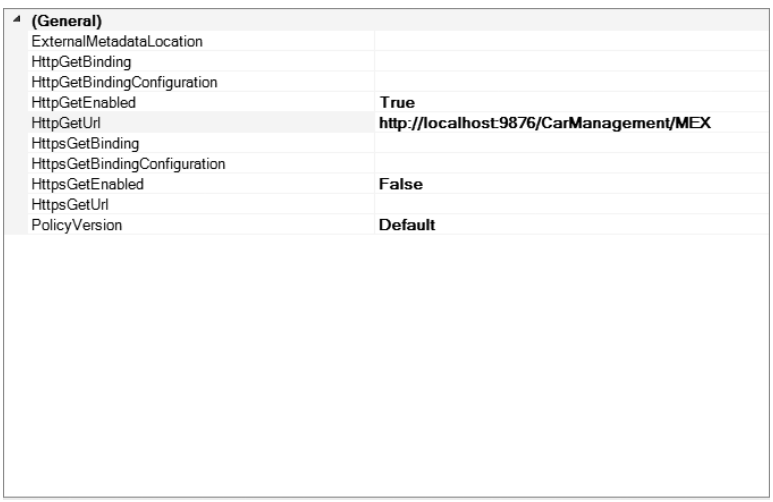


Рис. 11.36. Ввод `http://localhost:9876/CarManagement/MEX` в поле `HttpGetUrl`

Теперь обратимся к CarManagementService. Щелкните в древовидном представлении на этом элементе и установите BehaviorConfiguration равным ExposeMetaDataBehavior (рис. 11.37).

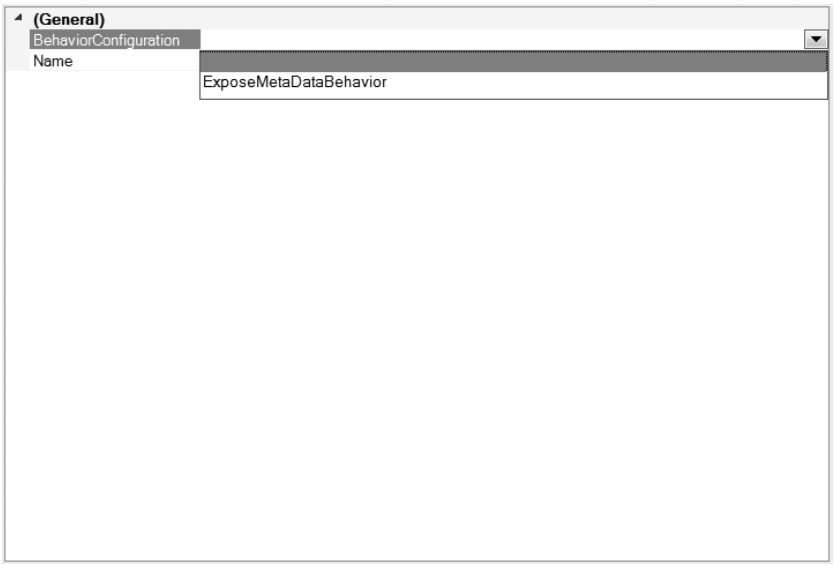


Рис. 11.37. Установка BehaviorConfiguration равным ExposeMetaDataBehavior

Сохраните конфигурацию, закройте WCF Configuration Editor и перезагрузите файл App.config.

Конфигурация в App.config сейчас включает дескриптор serviceBehavior.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="ExposeMetaDataBehavior">
      <serviceMetadata
       httpGetEnabled="true"
       httpGetUrl="http://localhost:9876/CarManagement/MEX" />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Конфигурация CarManagementService должна теперь выглядеть примерно так, как показано ниже (обратите внимание на атрибут behaviorConfiguration дескриптора service).

```
<service
    behaviorConfiguration="ExposeMetaDataBehavior"
    name="CarManagementService.CarManagementImplementation">
  <endpoint
      address="http://localhost:9876/CarManagementService"
      binding="wsHttpBinding"
      bindingConfiguration=""
      contract="CarManagementInterface.ICarManagement" />
</service>
```

Теперь можно протестировать конечную точку метаданных. Запустите приложение `HostAllServices` и браузер. После того как приложение `HostAllServices` заработает, перейдите по адресу `http://localhost:9876/CarManagement/MEX`. Браузер должен отобразить файл WSDL, как показано на рис. 11.38.

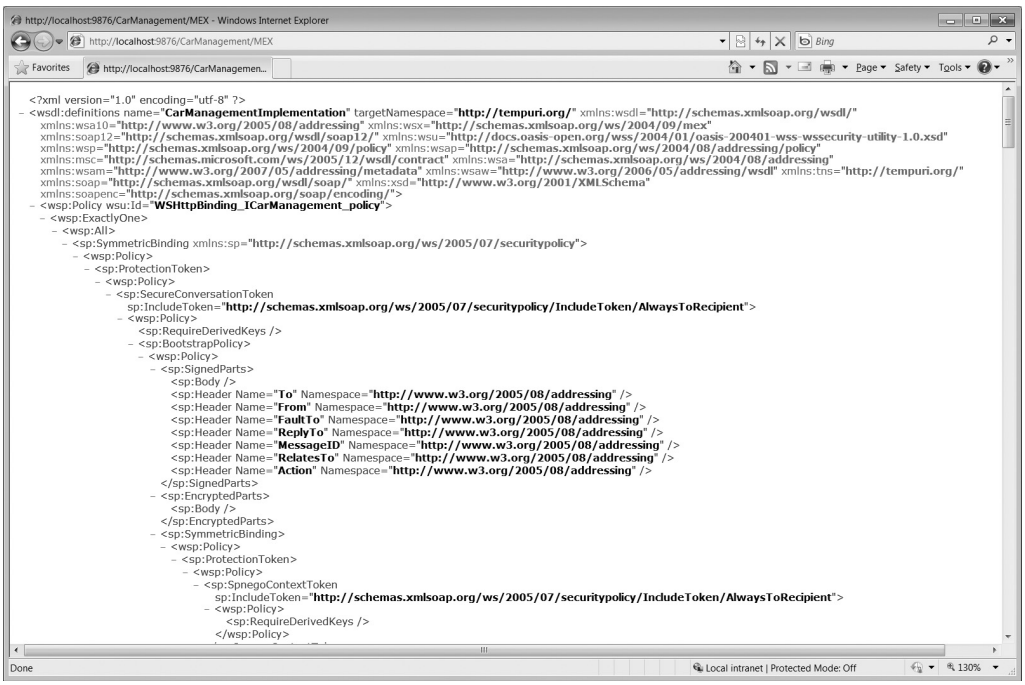


Рис. 11.38. Файл WSDL

Создание клиента `CarManagement`

Добавим в папку `Client` решения приложение Windows Forms с названием `CarApplication`. Это приложение использует службу `CarService`.

Запустите приложение `HostAllServices` извне Visual Studio (в предыдущем разделе вы запускали его из Visual Studio). Для этого перейдите в каталог проекта `bin\debug` и запустите `HostAllServices.exe`.

В Visual Studio щелкните на Add Service Reference в приложении `CarManagementClient`. Введите в диалоговом окне Add Service Reference адрес конечной точки метаданных — `http://localhost:9876/CarManagement/MEX`.

Установите пространство имен ссылки на службу `CarService` и щелкните на кнопке Go. Visual Studio загрузит WSDL и покажет доступные операции (рис. 11.39).

Щелкните на кнопке Advanced, чтобы открыть диалоговое окно Service Reference Settings. Установите тип Collection type равным `System.Collections.Generic.LinkedList` (рис. 11.40). Это необходимо для того, чтобы Visual Studio генерировал в прокси обобщенные списки вместо массивов для всех типов коллекций, обнаруженных в контракте.

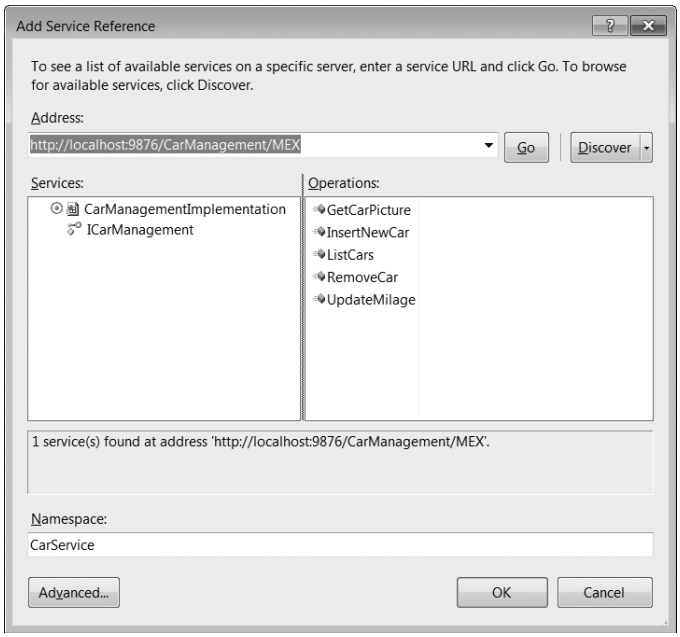


Рис. 11.39. Доступные операции

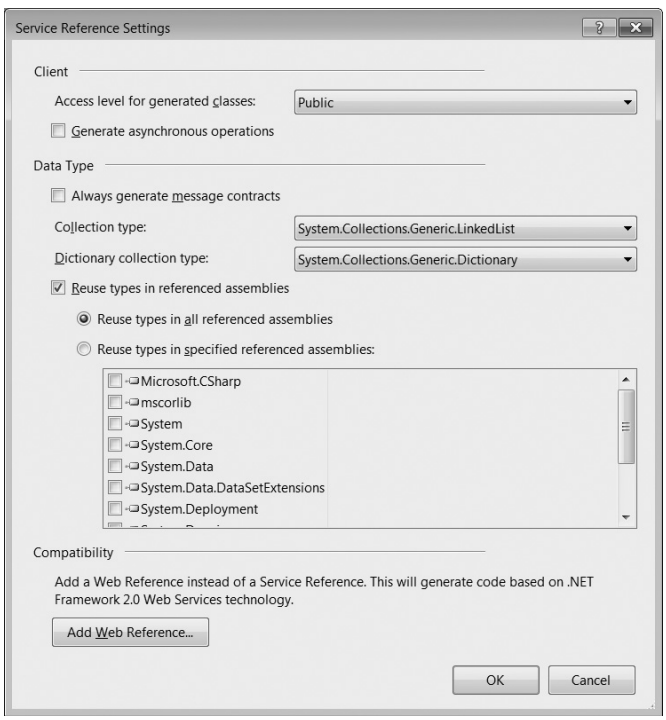


Рис. 11.40. Установка типа коллекции равным `System.Collections.Generic.LinkedList`

Дважды щелкните на кнопке OK, чтобы сгенерировать коды прокси и сконфигурировать клиентские конечные точки. В этом можно убедиться, щелкнув на кнопке Show All Files в Solution Explorer (рис. 11.41).

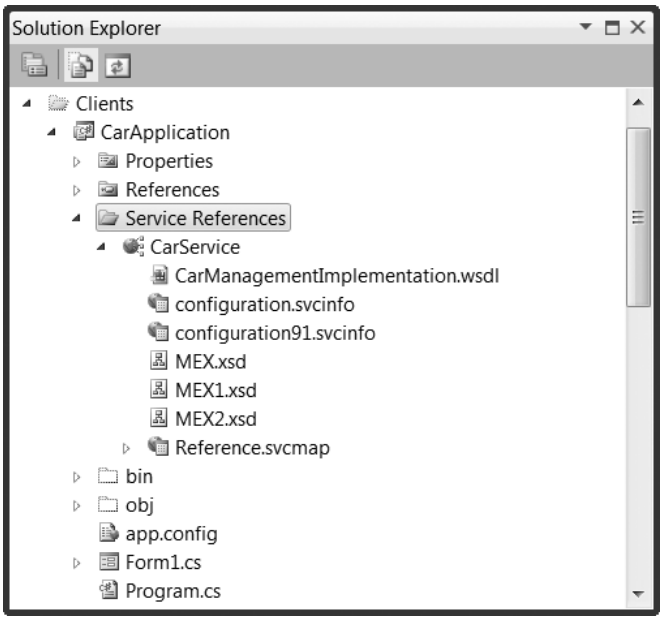


Рис. 11.41. Кнопка Show All Files в проводнике решений

Теперь можно закрыть приложение `HostAllServices` и просмотреть результат в Visual Studio. Откройте файл `App.config` и рассмотрите конфигурацию клиентской конечной точки. Для проверки закомментируйте дескриптор `<identity>` созданной клиентской точки. Результат должен выглядеть так, как показано ниже.

```
<client>
  <endpoint
    address="http://localhost:9876/CarManagementService"
    binding="wsHttpBinding"
    bindingConfiguration="WSHttpBinding_ICarManagement"
    contract="CarService.ICarManagement"
    name="WSHttpBinding_ICarManagement">
    <!-- <identity>
      <userPrincipalName value="XXX\YYY" />
    </identity> -->
  </endpoint>
</client>
```

Продолжим создание приложения, добавляя компоненты пользовательского интерфейса в форму. В данном случае нам нужны три кнопки: для списка, текстового поля и изображения. Форма должна выглядеть так, как показано на рис. 11.42.

Эти три кнопки вызывают операции `ListCars`, `InsertNewCar` и `GetCarPicture`. Список используется для вывода результатов, рисунок — для вывода изображения автомобиля, а многострочное текстовое поле — для вывода сообщений об ошибках, если таковые возникнут.

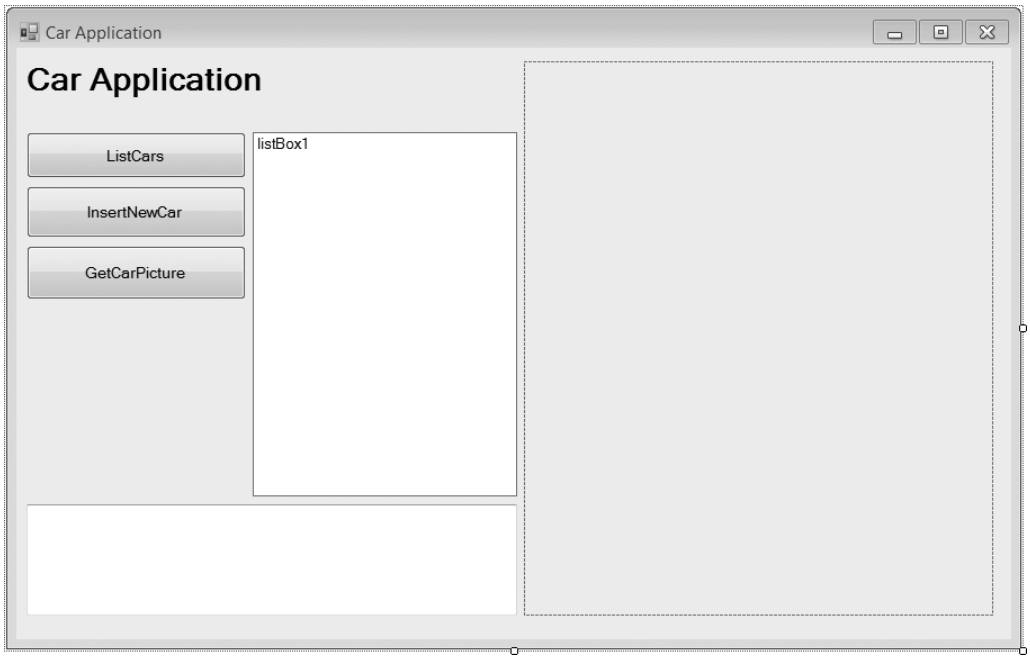


Рис. 11.42. Внешний вид формы

Добавьте следующий код к кнопке `ListCars`.

```
try
{
    CarService.CarManagementClient client;
    client = new CarService.CarManagementClient();
    List < CarService.Car > listCars;
    listCars = client.ListCars();
    foreach (CarService.Car car in listCars)
    {
        listBox1.Items.Add(car.BrandName +
                           " " + car.TypeName);
    }
}
catch (Exception ex)
{
    textBox1.Text = ex.Message;
}
```

Этот код инстанцирует прокси `CarManagementClient`, вызывает операцию `ListCars` и добавляет все возвращаемые автомобили в список.

Добавьте следующий код к кнопке `InsertNewCar`.

```
try
{
    CarService.CarManagementClient client;
    client = new CarService.CarManagementClient();
    CarService.Car car;
    car = new CarApplication.CarService.Car();
```

```
    car.BrandName = "BMW";
    car.TypeName = "320d"
    int newCarID;
    newCarID = client.InsertNewCar(car);
}
catch (Exception ex)
{
    textBox1.Text = ex.Message;
}
```

Этот код создает новый автомобиль, определяет его свойства и вызывает операцию службы `InsertNewCar`.

Добавьте следующий код к кнопке `GetCarPicture`.

```
try
{
    CarService.CarManagementClient client;
    client = new CarService.CarManagementClient();
    byte[] buff;
    buff = client.GetCarPicture("C67872");
    TypeConverter typeConverter;
    typeConverter =
        TypeDescriptor.GetConverter(typeof(Bitmap));
    Bitmap bitmap =
        (Bitmap)typeConverter.ConvertFrom(buff);
    pictureBox1.Image = bitmap;
}
catch (Exception ex)
{
    textBox1.Text = ex.Message;
}
```

Данный код вызывает операцию `GetCarPicture`, получает массив байтов, преобразует их в изображение и выводит его.

Необходимо также внести изменения в конфигурацию хоста, чтобы позволить ему корректно получать изображения. Чтобы получить от службы изображение, размер которого больше, чем размер, заданный по умолчанию, добавьте спецификацию привязки в файл `App.config` приложения `HostAllServices` и установите для привязки свойство `maxReceivedMessageSize`.

Откройте файл `App.config` и добавьте приведенный ниже код в дескриптор `system.serviceModel`.

```
<system.serviceModel>
  <bindings>
    <wsHttpBinding>
      <binding
            name="AllowBigMessageSize"
            maxReceivedMessageSize="999999">
      </binding>
    </wsHttpBinding>
  </bindings>
</system.serviceModel>
```

Здесь добавляется конфигурация привязки `AllowBigMessageSize`. Используйте эту конфигурацию в конфигурации конечной точки `CarManagementService`. Ниже показано, как выглядит код конфигурации.

```
<service
    behaviorConfiguration="ExposeMetaDataBehavior"
    name="CarManagementService.CarManagementImplementation">
  <endpoint
        address="http://localhost:9876/CarManagementService"
        binding="wsHttpBinding"
        bindingConfiguration="AllowBigMessageSize"
        contract="CarManagementInterface.ICarManagement" />
</service>
```

Измените также конфигурацию клиента, чтобы он мог получать большие сообщения. Откройте файл `App.config` для `CarApplication` и установите значение `maxReceivedMessageSize` для `WsHttpBinding` равным `999999`, а `maxArrayLength` для `readerQuotas` равным `999999`. Этого должно хватить для всех изображений.

```
<wsHttpBinding>
  <binding
    name="WSHttpBinding_ICarManagement"
    closeTimeout="00:01:00"
    openTimeout="00:01:00"
    receiveTimeout="00:10:00"
    sendTimeout="00:01:00"
    bypassProxyOnLocal="false"
    transactionFlow="false"
    hostNameComparisonMode="StrongWildcard"
    maxBufferPoolSize="524288"
    maxReceivedMessageSize="999999"
    messageEncoding="Text"
    textEncoding="utf-8"
    useDefaultWebProxy="true"
    allowCookies="false">
    <readerQuotas
      maxDepth="32"
      maxStringContentLength="8192"
      maxArrayLength="999999"
      maxBytesPerRead="4096"
      maxNameTableCharCount="16384" />
    <reliableSession
      ordered="true"
      inactivityTimeout="00:10:00"
      enabled="false" />
    <security mode="Message">
      <transport
        clientCredentialType="Windows"
        proxyCredentialType="None"
        realm="" />
      <message
        clientCredentialType="Windows"
        negotiateServiceCredential="true"
        algorithmSuite="Default"
        establishSecurityContext="true" />
    </security>
  </binding>
</wsHttpBinding>
```

Теперь можно протестировать и клиента, и службу. Чтобы запустить в Visual Studio оба проекта, измените свойства решения так, чтобы в нем имелось несколько стартовых проектов. Щелкните правой кнопкой мыши на решении и выберите в контекстном меню пункт Properties (рис. 11.43). Выберите переключатель Multiple startup projects и установите в поле Action значения для `CarApplication` и `HostAllServices` равными `Start`.

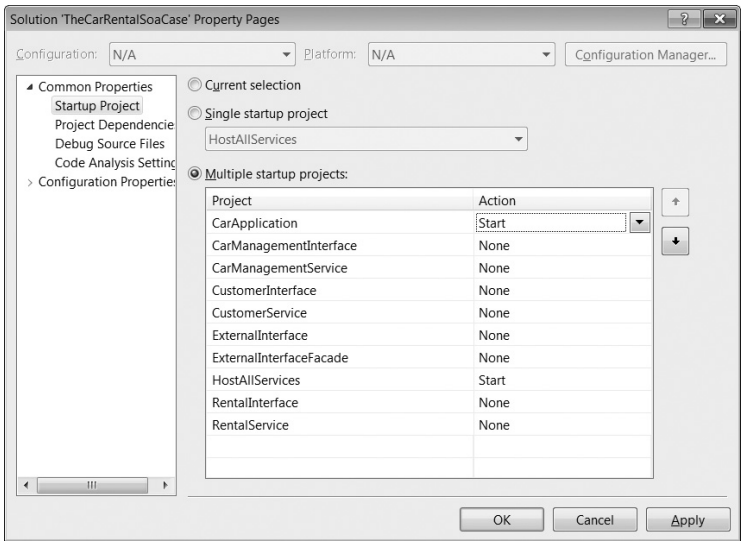


Рис. 11.43. Свойства проекта

Запустите приложение (запустив на выполнение все решение) и выполните операции, щелкая на кнопках (рис. 11.44). Взгляните на консоль приложения `ServiceHost`, чтобы увидеть, как вызываются эти операции.

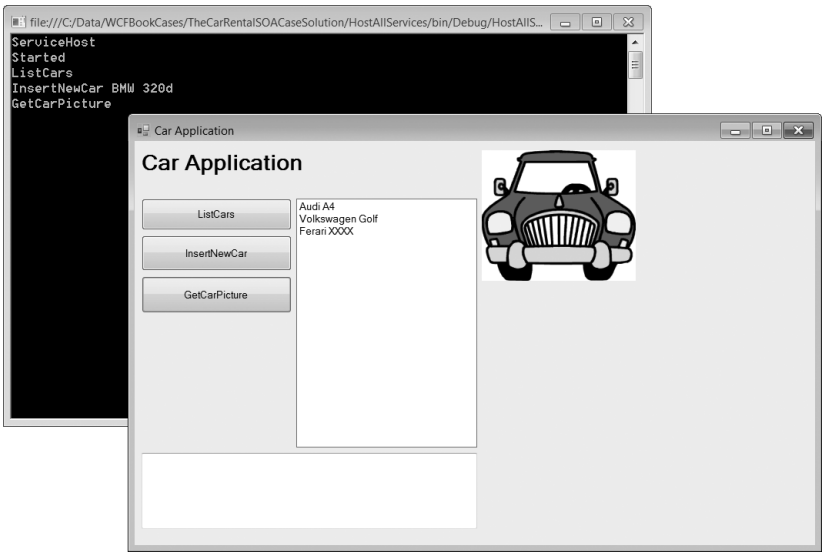


Рис. 11.44. Форма и консоль приложения `ServiceHost`

Создание RentalApplication

В папку Client решения добавим RentalApplication — приложение типа Windows form — и добавим к нему сборки System.ServiceModel и System.Runtime.Serialization.

Для этого клиента мы не будем создавать прокси путем добавления ссылки на службу в WSDL. Вместо этого обратимся к проекту RentalInterface, который имеет все необходимые метаданные в форме атрибутов WCF. Этот подход более простой и более гибкий, но годится только в том случае, когда клиент также представляет собой среду .NET. Для клиентов, не имеющих отношения к .NET, требуется файл WSDL.

Добавьте в приложение ссылку на RentalInterface (рис. 11.45).

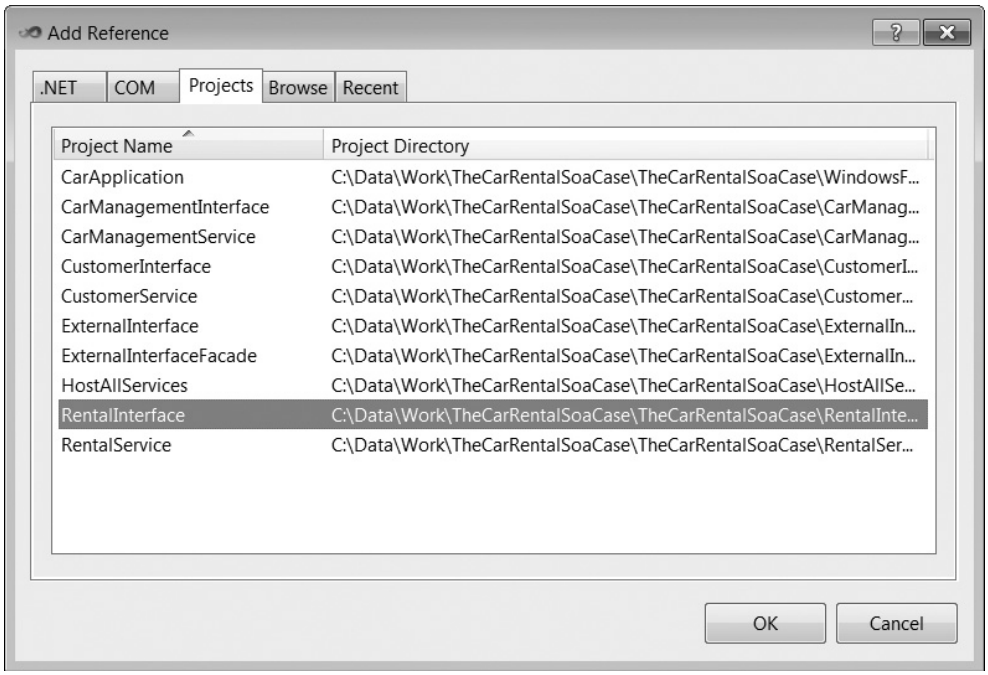


Рис. 11.45. Добавление в приложение ссылки на RentalInterface

Поскольку прокси создаются вручную, нам надо добавить в приложение класс RentalProxy, который представляет собой прокси, создаваемый вручную, а не генерируемый Visual Studio.

Добавьте в класс следующие инструкции using.

```
using System.ServiceModel;
using System.Runtime.Serialization;
```

Добавьте также инструкцию using для RentalInterface.

```
using RentalInterface;
```

Сделайте класс открытым, наследуйте его от ClientBase<RentalInterface.IRental> и реализуйте в нем интерфейс IRental. Код должен иметь следующий вид.

```
public class RentalProxy : ClientBase < IRental > ,IRental
{
    #region IRental Members
    public string RegisterCarRental(RentalRegistration
                                    rentalRegistration)
    {
        throw new NotImplementedException();
    }
    public void RegisterCarRentalAsPaid(string rentalID)
    {
        throw new NotImplementedException();
    }
    public void StartCarRental(string rentalID)
    {
        throw new NotImplementedException();
    }
    public void StopCarRental(string rentalID)
    {
        throw new NotImplementedException();
    }
    public RentalInterface.RentalRegistration
            GetRentalRegistration(string rentalID)
    {
        throw new NotImplementedException();
    }
    #endregion
}
```

Теперь следует создать конструктор класса, который вызывает конструктор базового класса со строкой в качестве параметра. Эта строка представляет собой имя конечной точки (ее мы создадим позже) в конфигурационном файле.

```
public RentalProxy()
    : base("RentalServiceEndpoint")
{
}
```

Реализуйте каждый метод с помощью кода, который вызывает соответствующий метод свойства канала. Свойство канала представляет собой защищенный член класса `ClientBase<T>` с использованным в объявлении класса обобщенным типом. В силу защищенности он доступен только в классах, являющихся потомками класса, в котором он был объявлен. В данном случае это интерфейс `IRental`. В результате это приводит к наличию в канале тех же методов, что и у класса реализации. Как видите, прокси просто представляет собой класс на стороне клиента с теми же методами, что и у класса на стороне службы. Следовательно, этот класс реализует интерфейс. Прокси просто передает получаемые от вызывающего кода параметры каналу и возвращает полученный из канала ответ обратно вызывающему коду.

После реализации интерфейса удалите инструкцию `throw` из каждого метода и замените ее следующим образом.

```
public string RegisterCarRental(RentalRegistration rental)
{
    return Channel.RegisterCarRental(rental);
}
```

```
public void RegisterCarRentalAsPaid(string rentalID)
{
    Channel.RegisterCarRentalAsPaid(rentalID);
}
public void StartCarRental(string rentalID)
{
    Channel.StartCarRental(rentalID);
}
public void StopCarRental(string rentalID)
{
    Channel.StopCarRental(rentalID);
}
public RentalRegistration
    GetRentalRegistration(string rentalID)
{
    return Channel.GetRentalRegistration(rentalID);
}
```

Поскольку мы отказались от подхода с добавлением ссылки на службу, вам надо добавить и сконфигурировать файл `App.config` самостоятельно. Добавьте файл `App.config` в приложение и отредактируйте его с помощью WCF Configuration Editor (рис. 11.46).

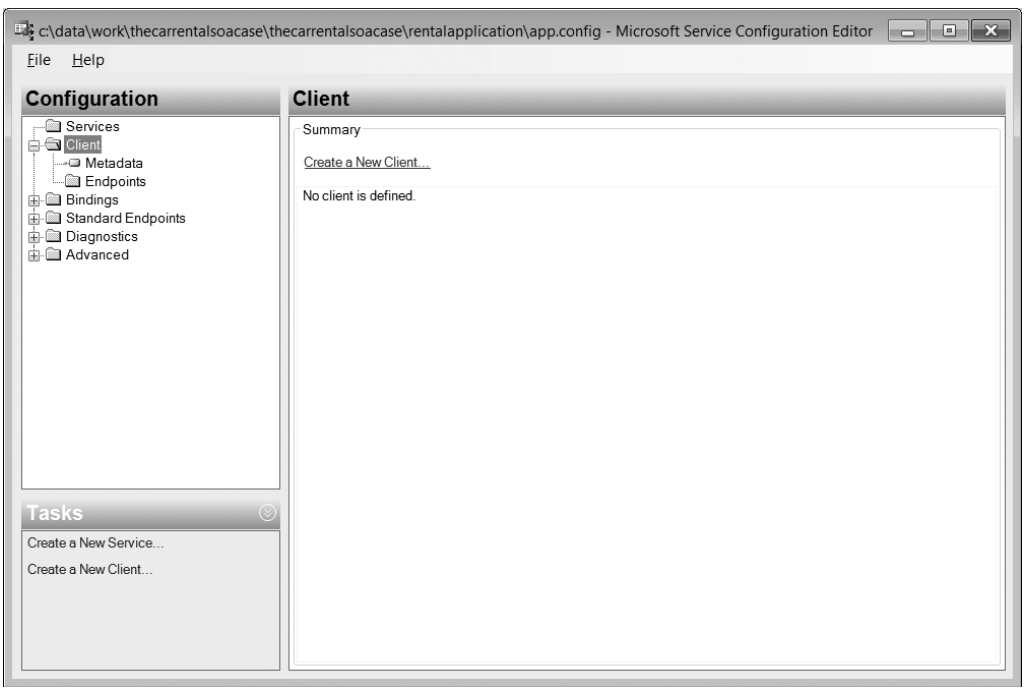


Рис. 11.46. Редактирование файла `App.config`

Добавьте конфигурацию клиента щелчком на ссылке Create a New Client. После запуска мастер запросит о том, как следует создать конфигурацию (рис. 11.47).

Рис. 11.47. Запрос о создании конфигурации

Выберите в диалоговом окне вариант From Service Config и щелкните на кнопке Browse, чтобы отыскать файл `App.config` приложения `HostAllServices`. Это файл `C:\Data\Work\TheCarRentalSOACase\TheCarRentalSOACase\HostAllServices\app.config`.

Выберите конечную точку для подключения клиента. В раскрывающемся списке имеются все четыре конечные точки (рис. 11.48). Выберите конечную точку `RentalService`.

Рис. 11.48. Выбор конечной точки для подключения клиента

Теперь мастер запрашивает имя, идентифицирующее конфигурацию клиента (`RentalServiceEnpoint`) (рис. 11.49). Это та строка, которая указывается в конструкторе прокси, и благодаря ей WCF знает, где искать конфигурацию при инстанцировании прокси.

Рис. 11.49. Имя, идентифицирующее конфигурацию клиента

Щелкните на кнопках Next и Finish, сохраните конфигурацию, закройте WCF Configuration Editor и перезагрузите файл `App.config` в Visual Studio.

Для тестирования удалите дескриптор `<identity>` в сгенерированной конфигурации, которая должна иметь следующий вид.

```
<system.serviceModel>
  <client>
    <endpoint
        address="http://localhost:9876/RentalService"
        binding="wsHttpBinding"
        bindingConfiguration=""
        contract="RentalInterface.IRental"
        name="RentalServiceEndpoint">
    </endpoint>
  </client>
</system.serviceModel>
```

Для того чтобы создать пользовательский интерфейс, добавьте в форму две кнопки (рис. 11.50). Эти кнопки будут выполнять код, вызывающий прокси.

Рис. 11.50. Две новые кнопки в форме

Вот как выглядит код приложения RegisterCarRental.

```
try
{
    RentalProxy rentalProxy;
    rentalProxy = new RentalProxy();
    RentalInterface.RentalRegistration rentalRegistration;
    rentalRegistration = new RentalInterface.RentalRegistration();
    rentalRegistration.CustomerID = 1;
    rentalRegistration.CarID = "123767";

    rentalRegistration.DropOffLocation = 1327;
    rentalRegistration.DropOffDateTime = System.DateTime.Now;

    rentalRegistration.PickUpLocation = 7633;
    rentalRegistration.PickUpDateTime = System.DateTime.Now;

    rentalProxy.RegisterCarRental(rentalRegistration);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

А ниже приведен код для кнопки RegisterCarRentalAsPaid.

```
try
{
    RentalProxy rentalProxy;
    rentalProxy = new RentalProxy();

    rentalProxy.RegisterCarRentalAsPaid("1876893");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Для тестирования измените свойства решения, установив в качестве стартовых приложения RentalApplication и HostAllServices. Запустите приложения, вызовите операции щелчками на соответствующих кнопках, после чего проверьте корректность внесения информации в базу данных.

Добавление обработки ошибок

В настоящее время наши приложения не содержат корректной обработки ошибок. Хорошая стратегия обработки ошибок начинается с определения соответствующих контрактов в интерфейсах.

Откройте файл IRental.cs и добавьте контракт.

```
[DataContract(Name = "RentalRegisterFault",
              Namespace="FaultContracts/RentalRegisterFault")]
public class RentalRegisterFault
{
    [DataMember]
    public string FaultDescription { get; set; }
    [DataMember]
    public int FaultID { get; set; }
}
```

Теперь добавьте атрибут `FaultContract` к каждому методу интерфейса `IRental`. Интерфейс примет следующий вид.

```
[ServiceContract]
public interface IRental
{
    [OperationContract]
    [FaultContract(typeof(RentalRegisterFault))]
    string RegisterCarRental(RentalRegistration
                             rentalRegistration);

    [OperationContract]
    [FaultContract(typeof(RentalRegisterFault))]
    void RegisterCarRentalAsPaid(string rentalID);

    [OperationContract]
    [FaultContract(typeof(RentalRegisterFault))]
    void StartCarRental(string rentalID);

    [OperationContract]
    [FaultContract(typeof(RentalRegisterFault))]
    void StopCarRental(string rentalID);

    [OperationContract]
    [FaultContract(typeof(RentalRegisterFault))]
    RentalRegistration
            GetRentalRegistration(string rentalID);
}
```

Следующий шаг состоит в добавлении логики для обнаружения исключений в реализации службы. Это делается на двух уровнях. Первый уровень состоит в проверке значения входного параметра на равенство нулю. В случае нулевого значения передается исключение `FaultException` типа `RentalRegisterFault`. Код вставки информации о договоре аренды окружается блоком `try...catch`. В случае ошибки в обработчике передается исключение `FaultException` типа `RentalRegisterFault`.

Ниже приведен измененный код реализации.


```
public string RegisterCarRental(RentalRegistration
                                rentalRegistration)
{
    Console.WriteLine("RegisterCarRental");
    if (rentalRegistration == null)
    {
        RentalRegisterFault fault;
        fault = new RentalRegisterFault();
        fault.FaultID = 1;
        fault.FaultDescription =
            "Input is not valid, got null value";
        throw new
            FaultException<RentalRegisterFault>(fault,"");
    }

    try
```

```
    {
        using (DataClassesRentalDataContext ctx =
            new DataClassesRentalDataContext())
        {
            Rental rentalToInsert;
            rentalToInsert = new Rental();
            rentalToInsert.CustomerID =
                rentalRegistration.CustomerID;
            rentalToInsert.CarID = rentalRegistration.CarID;
            rentalToInsert.Comments =
                rentalRegistration.Comments;
            ctx.Rentals.InsertOnSubmit(rentalToInsert);
            ctx.SubmitChanges();
            return "OK";
        }
    }
    catch (Exception ex)
    {
        RentalRegisterFault fault;
        fault = new RentalRegisterFault();
        fault.FaultID = 123;
        fault.FaultDescription =
            "An error occurred while "
            "inserting the registration";
        throw new
            FaultException<RentalRegisterFault>(fault,"");
    }
}
```

Файл CreatingaSOACase.zip

На стороне клиента разумно перехватывать четыре типа исключений.

- `FaultException` типа `RentalRegisterFault`.
- Обобщенное исключение `FaultException`.
- `EndpointNotFoundException`.
- `CommunicationException`.

Модифицированный код кнопки в клиентском приложении приведен ниже.


```
try
{
    RentalProxy rentalProxy;
    rentalProxy = new RentalProxy();

    RentalInterface.RentalRegistration rentalRegistration;
    rentalRegistration =
        new RentalInterface.RentalRegistration();
    rentalRegistration.CustomerID = 1;
    rentalRegistration.CarID = "123767";

    rentalRegistration.DropOffLocation = 1327;
    rentalRegistration.DropOffDateTime =
        System.DateTime.Now;
```

```
    rentalRegistration.PickUpLocation = 7633;
    rentalRegistration.PickUpDateTime = System.DateTime.Now;

    rentalProxy.RegisterCarRental(rentalRegistration);
}
catch (FaultException<RentalInterface.RentalRegisterFault>
        rentalRegisterFault)
{
    MessageBox.Show("rentalRegisterFault " +
                    rentalRegisterFault.Message);
}
catch (FaultException faultException)
{
    MessageBox.Show("faultException " +
                    faultException.Message);
}
catch (EndpointNotFoundException endpointNotFoundException)
{
    MessageBox.Show("endpointNotFoundExc " +
                    endpointNotFoundException.Message);
}
catch (CommunicationException communicationException)
{
    MessageBox.Show("communicationException" +
                    communicationException.Message);
}
```

Файл CreatingaSOACase.zip

Имперсонация клиента

Еще одно требование состояло в том, что вызов метода `RegisterCarRentalAsPaid` в `RentalService` должен быть имперсонирован. Это означает, что код реализации в службе должен запускаться от имени учетных данных пользователя клиентского приложения.

Откройте файл `RentalServiceImplementation.cs` и добавьте в него инструкцию `using` для пространства имен `System.Security.Principal`. Тем самым делается доступным класс `WindowsIdentity`, который позволяет получить имя пользователя, запустившего код. Таким образом, можно протестировать, действительно ли имперсонация работает корректно.

```
using System.Security.Principal;
```

Необходимо также в реализации добавить атрибут `OperationBehavior` к методу `RegisterCarRentalAsPaid` и установить параметр имперсонации атрибута равным `Required`.

```
[OperationBehavior(Impersonation = ImpersonationOption.Required)]
```

Полностью метод принимает следующий вид.

```
[OperationBehavior(Impersonation =
                   ImpersonationOption.Required)]
public void RegisterCarRentalAsPaid(string rentalID)
{
    Console.WriteLine("RegisterCarRentalAsPaid " +
```

```
                    rentalID);
    Console.WriteLine(" WindowsIdentity : {0} ",
                    WindowsIdentity.GetCurrent().Name);
}
```

Проверку можно выполнить, запустив `RentalApplication` из командной строки с помощью утилиты `runas`. Данная утилита позволяет указать, что приложение запускает не тот пользователь, который вошел в систему. Для этого надо открыть окно командной строки, перейти в каталог, в котором находится `RentalApplication`, и выполнить команду

```
runas /user:UserX RentalApplication.exe
```

Ею вы указываете, что приложение `RentalApplication.exe` запускается от имени пользователя `UserX`. Можете использовать любое имя пользователя из вашего домена, чей пароль вам известен. Утилита `runas` запрашивает пароль указанного пользователя, а затем запускает приложение.

Расширение интерфейса `CarManagement` для подтипов автомобилей

Еще одно требование обеспечивает возможность изменять подкласс класса автомобиля. Добавьте два класса, которые наследуют класс `Car`, к интерфейсу `ICarManagement`.

```
[DataContract]
public class LuxuryCar : Car
{
    [DataMember]
    List <LuxuryItems> LuxuryItemsList { get; set; }
}

[DataContract]
public class LuxuryItems
{
    [DataMember]
    public string ItemName { get; set; }
    [DataMember]
    public string ItemDescription { get; set; }
}

[DataContract]
public class SportsCar : Car
{
    [DataMember]
    public int HorsePower { get; set; }
}
```

Определим атрибут `KnownType` класса `Car` для каждого возможного подкласса. Применение этого атрибута позволяет возвращаемым значениям метода `ListCars` быть типами, унаследованными от класса `Car`. В данном случае это типы `LuxuryCar` и `SportsCar`.

```
[DataContract]
[KnownType(typeof(LuxuryCar))]
```

```
[KnownType(typeof(SportsCar))]
public class Car
{
    [DataMember]
    public string BrandName { get; set; }
    [DataMember]
    public string TypeName { get; set; }
    [DataMember]
    public TransmissionTypeEnum Transmission { get; set; }
    [DataMember]
    public int NumberOfDoors { get; set; }
    [DataMember]
    public int MaxNumberOfPersons { get; set; }
    [DataMember]
    public int LitersOfLuggage { get; set; }
}
```

Теперь можно изменить реализацию метода `ListCars` таким образом, чтобы он включал в возвращаемый список `SportsCar`.

```
public List < Car > ListCars()
{
    Console.WriteLine("ListCars");
    List < Car > listCars;
    listCars = new List < Car > ();
    listCars.Add(new Car {
        BrandName = "Audi",
        Transmission = TransmissionTypeEnum.Automatic,
        TypeName = "A4" });
    listCars.Add(new Car {
        BrandName = "Volkswagen",
        Transmission = TransmissionTypeEnum.Automatic,
        TypeName = "Golf" });
    listCars.Add(new SportsCar {
        BrandName = "Ferrari",
        Transmission = TransmissionTypeEnum.Automatic,
        TypeName = "XXXX", HorsePower= 600 });
    return listCars;
}
```

Использование атрибута `KnownType` приводит к новой версии контракта, а равно к новой версии файла WSDL. Этот новый файл WSDL теперь также включает структуру двух типов, унаследованных от типа `Car`. Теперь нам надо обновить клиентов с применением этого контракта.

Чтобы обновить ссылку на службу в `CarApplication`, вновь запустите приложение `HostAllServices` извне Visual Studio. Щелкните правой кнопкой мыши на существующей службе `CarService` и выберите Update Service Reference (рис. 11.51).

Это приводит к генерации прокси, и теперь два рассмотренных подтипа будут известны клиенту.

Чтобы убедиться в этом, закройте приложение `HostAllServices` и замените стартовый проект в решении приложениями `CarApplication` и `HostAllServices`. После этого вновь попытайтесь получить список автомобилей. Теперь результат должен включать автомобили одного из подтипов.

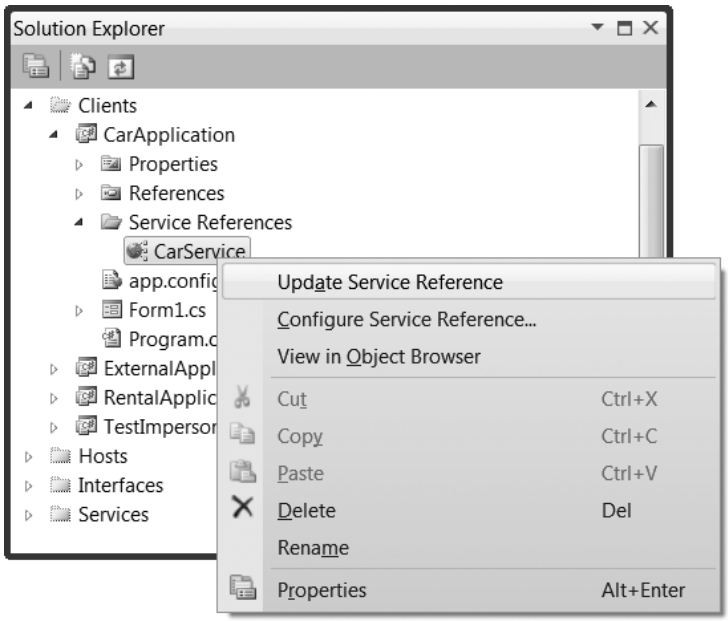


Рис. 11.51. Обновление ссылки на службу

Реализация `ExternalInterfaceFacade`

Реализация `ExternalInterface` имеет один метод, который называется `SubmitRentalContract` и получает параметр, содержащий данные о новом пользователе и договоре аренды. Назначение метода `SubmitRentalContract` — внесение нового пользователя и договора аренды в единой транзакции. Данный метод вызывает метод `RegisterCustomer` из `CustomerService` и метод `RegisterCarRental` из `RentalService`.

Использование `ExternalInterfaceFacade`

`ExternalInterfaceFacade` вызывает две указанные службы с помощью привязки к именованным каналам. Осуществляется это путем создания канала к службам с помощью `ChannelFactory` и указания типа привязки и ее адреса в коде, а не в конфигурации. Это еще один способ создания прокси. Вместо того чтобы добавлять ссылку на службу или создавать прокси, наследующий класс `ClientBase`, вручную, воспользуемся `ChannelFactory` для динамического создания прокси во время выполнения программы на основе интерфейса.

Определение области видимости транзакции выполняется с помощью блока `using`, в котором инстанцируется `TransactionScope`. После вызова обоих методов область видимости помечается как завершенная.

Добавьте в проект `ExternalInterfaceFacade` ссылку на `System.Transactions` (рис. 11.52).

Откройте файл `ExternalInterfaceFacadeImplementation.cs` и добавьте инструкцию `using` для пространства имен `System.Transactions`:

```
using System.Transactions;
```

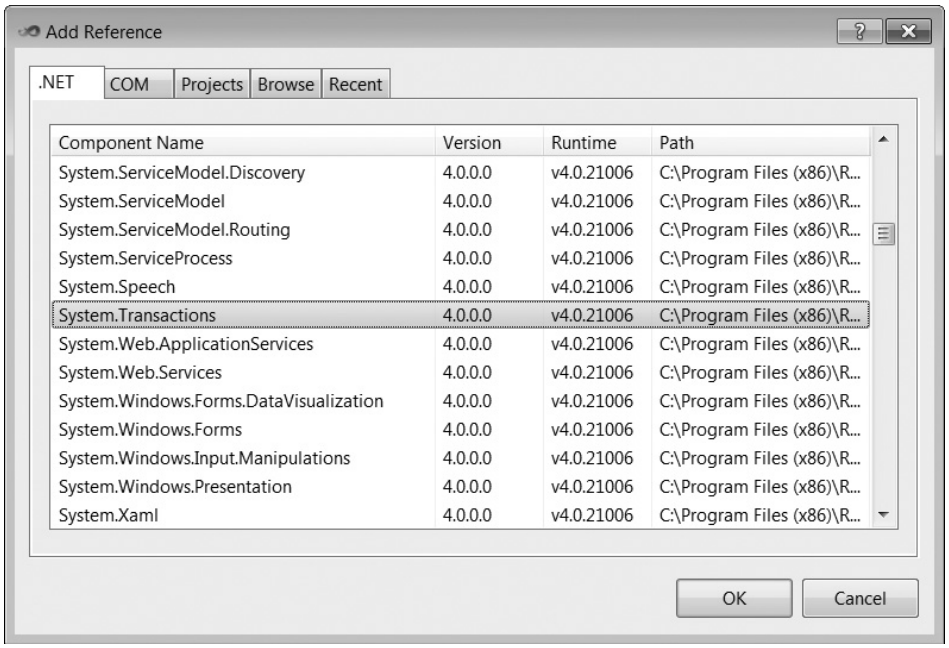


Рис. 11.52. Добавление в проект `ExternalInterfaceFacade` *ссылки на* `System.Transactions`

Добавьте в проект `ExternalInterfaceFacade` ссылки на библиотеки `CustomerInterface` и `RentalInterface` (рис. 11.53).

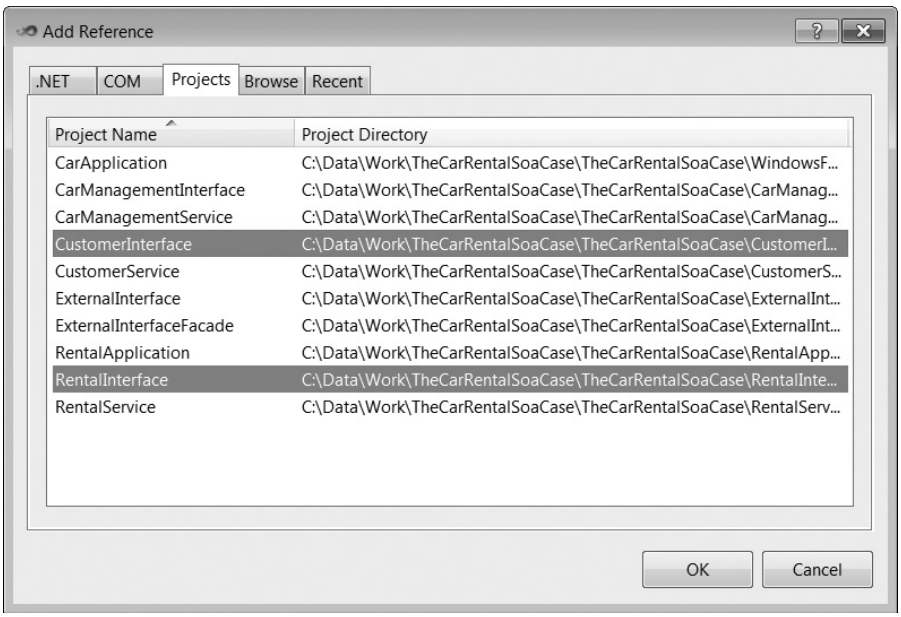


Рис. 11.53. Добавление в проект `ExternalInterfaceFacade` *ссылок на библиотеки*

Добавьте следующий код в реализацию метода SubmitRentalContract.

```
using (TransactionScope scope =
    new TransactionScope(TransactionScopeOption.RequiresNew))
{
    NetNamedPipeBinding netNamedPipeBinding;
    netNamedPipeBinding = new NetNamedPipeBinding();
    netNamedPipeBinding.TransactionFlow = true;

    CustomerInterface.ICustomer customerServiceChannel;
    customerServiceChannel =
        ChannelFactory<CustomerInterface.ICustomer>.
        CreateChannel(netNamedPipeBinding, new
            EndpointAddress(
                "net.pipe://localhost/customerservice"));

    int newCustomerID;
    newCustomerID =
        customerServiceChannel.RegisterCustomer(
            rentalContract.Customer);
    rentalContract.RentalRegistration.CustomerID =
        newCustomerID;

    RentalInterface.IRental rentalServiceChannel;
    rentalServiceChannel =
        ChannelFactory<RentalInterface.IRental>.
        CreateChannel(netNamedPipeBinding, new
            EndpointAddress(
                "net.pipe://localhost/rentalservice"));

    rentalServiceChannel.RegisterCarRental(
        rentalContract.RentalRegistration);

    scope.Complete();
}
```

В данном коде транзакция начинается в блоке using. В нем инстанцируется привязка netNamedPipeBinding с флагом transactionFlow, равным true, которая затем используется для создания двух каналов — одного для CustomerService и второго для RentalService. Первой вызывается операция RegisterCustomer из CustomerService, затем операция RegisterCarRental из RentalService. После вызова обеих операций транзакция завершается.

Поддержка транзакций на уровне методов

Теперь для двух методов надо указать, что для них требуется транзакционное поведение и что они могут автоматически завершать транзакцию.

Добавьте атрибут OperationBehavior к методу RegisterCustomer в реализации CustomerService. Установите значения TransactionAutoComplete и TransactionScopeRequired равными true.

```
[OperationBehavior(TransactionAutoComplete = true,
                   TransactionScopeRequired = true)]
public int RegisterCustomer(CustomerInterface.Customer customer)
{
    //...
}
```

Добавьте атрибут `OperationBehavior` к методу `RegisterCarRental` в реализации `RentalService`. Установите значения `TransactionAutoComplete` и `TransactionScopeRequired` равными `true`.

```
[OperationBehavior(TransactionAutoComplete = true,
                   TransactionScopeRequired = true)]
public string RegisterCarRental(RentalRegistration rentalRegistration)
{
    //...
}
```

Настройка дополнительных конечных точек для хоста службы

Добавьте конечную точку `netNamedPipeBinding` в `CustomerService` в файл `App.config` приложения `HostAllServices`.

```
<service name="CustomerService.CustomerServiceImplementation">
...
  <endpoint
    address="net.pipe://localhost/customerservice"
    binding="netNamedPipeBinding"
    bindingConfiguration="SupportTransactionsNetNamedBinding"
    contract="CustomerInterface.ICustomer" />
</service>
```

Выполните те же действия для `RentalService`.

```
<service name="RentalService.RentalServiceImplementation">
...
  <endpoint
    address="net.pipe://localhost/rentalservice"
    binding="netNamedPipeBinding"
    bindingConfiguration="SupportTransactionsNetNamedBinding"
    contract="RentalInterface.IRental" />
</service>
```

Эти конечные точки ссылаются на конфигурацию `bindingConfiguration` с именем `SupportTransactionsNetNamedBinding`. Привязку надо настроить так, как показано ниже.

```
<bindings>
  <netNamedPipeBinding>
    <binding name="supportTransactionsNetNamedBinding"
             transactionFlow="true">
    </binding>
  </netNamedPipeBinding>
</bindings>
```

Вернитесь к коду и добавьте атрибут `TransactionFlow` к операции `RegisterCarRental` интерфейса `IRental` со значением `transactionFlowOption`, равным `Allowed`.

```
[OperationContract]
[FaultContract(typeof(RentalRegisterFault))]
[TransactionFlow(TransactionFlowOption.Allowed)]
string RegisterCarRental(RentalRegistration rentalRegistration);
```

Сделайте то же и для метода `RegisterCustomer` интерфейса `Icustomer`.

```
[OperationContract]
[TransactionFlow(TransactionFlowOption.Allowed)]
int RegisterCustomer(Customer customer);
```

12

Создание примера коммуникации и интеграции

В ЭТОЙ ГЛАВЕ...

- Настройка решения WCF для примера коммуникации и интеграции
- Настройка и работа с маршрутизацией WCF
- Шаблон издания/подписки
- Соединение с веб-службами ASMX
- Использование привязки MSMQ
- Реализация подхода на основе REST

Эта глава представляет собой интерактивное пошаговое руководство о том, как создать пример работы с коммуникациями и интеграцией в Visual Studio 2010 с применением WCF. Она начинается с определения требований к примеру и показывает, как шаг за шагом разрабатывается полное решение. В конце главы будут разработаны интерфейсы, хосты и клиенты, которые взаимодействуют друг с другом. Это завершенный пример, который можно проверить в действии и убедиться в его работоспособности.

Определение требований

Глобальная американская компания продает свою продукцию клиентам по всему миру. Заказы принимаются и вводятся в штаб-квартире в США, но товары поставляются из складов по всему миру (Бельгия, Аргентина и т.д.). Компания должна интегрировать существующие приложения в различных филиалах по всему миру для ввода заказов, их отслеживания и управления местной доставкой, причем для выполнения заказа информацией могут обмениваться различные типы клиентов и служб (как давно существующие, так и новые).

Заказ создается в штаб-квартире в США с использованием существующего приложения для ввода заказа. Это приложение не использует WCF, так как было разработано до того, как платформа WCF стала доступна. Оно использует очередь сообщений на основе MSMQ для отправки сообщений, содержащих данные для других приложений.

Эти данные должны быть получены службой, работающей в штаб-квартире в США, которая является ответственной за первые проверки корректности заказа. Эта проверка представляет собой функциональность, которая не может быть реализована старым клиентом ввода заказов. Заказ признается недействительным, если заказанные товары недоступны для страны, из которой поступил заказ. Данная служба определяет, какие товары доступны, обращаясь к службе для каждой страны.

Еще одна работа этой службы — добавление информации в сообщение. Поскольку приложение для ввода заказов было разработано до того, как компания начала бизнес в глобальном масштабе, формат идентификатора товара не поддерживает работу в глобальном масштабе. Служба добавляет префикс для таких идентификаторов, указывающий страну, в которой товар имеется на складе.

Проверка доступности товара в стране осуществляется еще одной службой — ASMX .NET 2.0, — которая выполняет поиск информации в небольшой базе данных SQL Server.

Служба также должна добавлять локализованные описания каждого заказанного товара, вызывая службу, которая располагается во внутренней сети (интранете), но не работает в качестве службы SOAP. Это служба на основе REST, которая выдает в ответ на запрос код XML, содержащий локализованное описание товара.

После того как данные проверены и признаны корректными, они пересылаются службам в соответствующее местное отделение. Это делается посредством службы маршрутизации, позволяющей новым филиалам в других странах быстро получать заказы. В каждом филиале службы имеют один и тот же интерфейс, но могут использовать различные коммуникационные протоколы — например, филиал в Аргентине использует `WSHttpBinding`, а отделение в Бельгии — `NetTcpBinding`. Это связано с настройкой брандмауэров, где Аргентина не имеет права использовать в качестве протокола `netTCP`.

Имеется еще одно приложение, которое отслеживает количество заказов по каждой стране. Данное приложение всегда работает и должно выводить эти показатели в реальном времени на рабочем столе генерального директора компании. Генеральный директор считает это важным показателем и хочет постоянно видеть его, не щелкая ни на каких кнопках. Это приложение подписано на события, поступающие от службы `OrderEntry`.

Полная схема системы представлена на рис. 12.1.

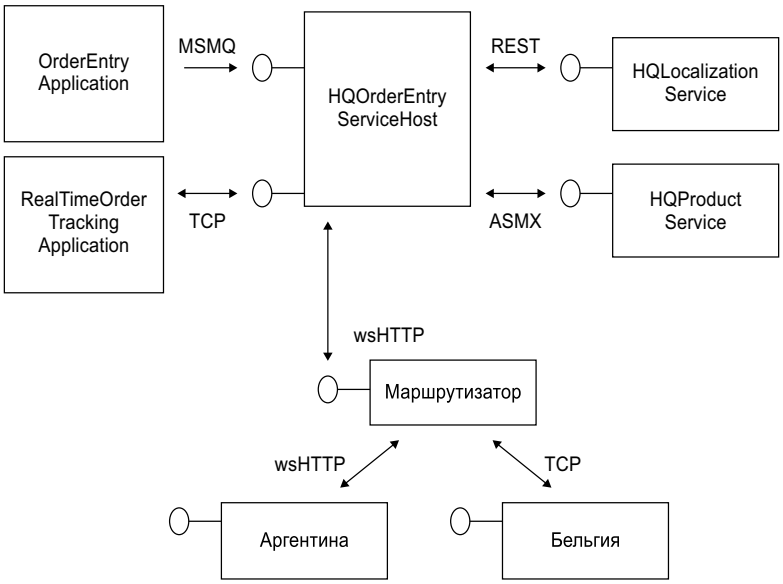


Рис. 12.1. Полная схема системы

Настройки решения

Полное решение состоит из тринадцати проектов, которые делятся на пять групп. В результате получаем следующий список проектов.

1. Службы
   - `HQOrderEntryServiceHost`. Консольное приложение, служащее хостом для размещения службы `OrderEntry`, работающей в штаб-квартире.
   - `HQOrderEntryImplementation`. Библиотека классов, содержащая реализацию операций по получению заказов; кроме того, обеспечивает подписку на отслеживаемую информацию.
   - `HQLocalizationService`. Консольное приложение, служащее хостом для размещения службы локализации, ответственной за перевод описания товаров. В этом приложении совместно присутствуют хост, интерфейс и реализация.
   - `HQProductServiceASMX`. Классическая веб-служба .NET 2.0 ASMX, отвечающая за проверку доступности товара в стране.
   - `RouterHost`. Консольное приложение, служащее для размещения службы, работающей в качестве маршрутизатора.
2. Интерфейсы служб
   - `HQOrderEntryServiceInterface`. Библиотека классов, содержащая интерфейс, описывающий операцию получения заказа.
   - `LocalOrderEntryInterface`. Библиотека классов, содержащая интерфейс для служб ввода заказов в филиалах.
   - `RealTimeOrderTrackingCallBackContract`. Библиотека классов, содержащая контракт для вызовов отслеживающего приложения из `HQOrderEntryServiceHost`.

3. Клиенты
   - `OrderEntryApplication`. Приложение Windows для ввода заказов. Отправляет заказ в очередь в виде сообщения.
   - `RealTimeOrderTrackingApplication`. Приложение Windows, показывающее количество заказов для каждой страны в реальном времени.
4. Филиалы
   - `BelgiumHost`. Консольное приложение, в качестве хоста предоставляющее реализацию `LocalOrderEntryInterface` для Бельгии с применением привязки `netTcpBinding`.
   - `ArgentinaHost`. Консольное приложение, в качестве хоста предоставляющее реализацию `LocalOrderEntryInterface` для Аргентины с применением привязки `wsHttpBinding`.
5. Вспомогательная библиотека
   - `HelperLib`. Библиотека классов, содержащая функции для сериализации и десериализации.

Начнем с создания пустого решения, `CommunicationAndIntegrationCase`, и добавим в проект пять папок решений (рис. 12.2).

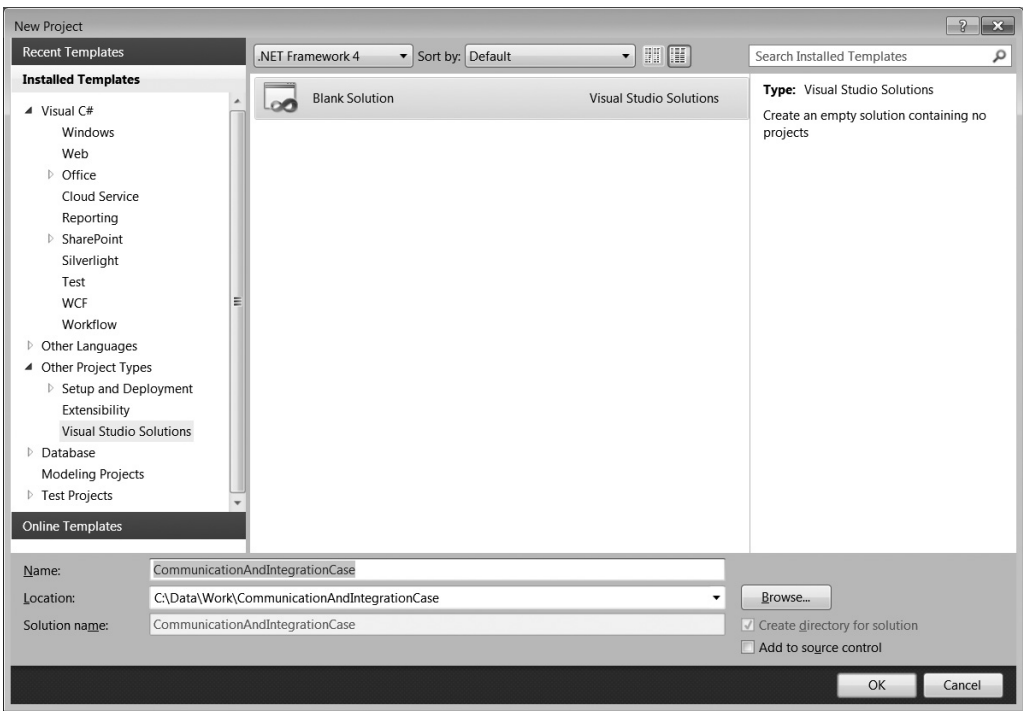


Рис. 12.2. Пустое решение

Эти папки показаны на рис. 12.3.

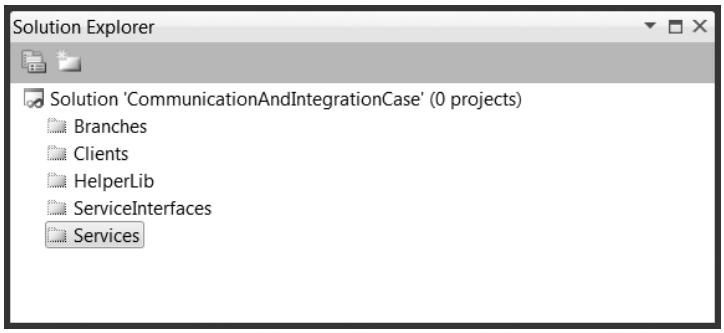


Рис. 12.3. Добавление папок

Создание HQOrderEntryInterface

Интерфейс для получения заказа — WCF `ServiceContract` — создается в отдельной библиотеке классов, так что к нему можно обращаться из множества приложений.

Добавьте проект библиотеки классов в папку решения `HQOrderEntryService-Interface`, а также ссылку на библиотеки `System.ServiceModel` и `System.Runtime.Serialization`.

Добавьте класс `HQOrderEntry`, который определяет структуру заказа как контракт службы WCF.

Добавьте инструкцию `using`, чтобы сделать доступными атрибуты из пространства имен `System.Runtime.Serialization`.

```
using System.Runtime.Serialization;
```

Теперь добавьте в класс `OrderEntry` следующий код.

```
[DataContract]
public class HQOrderEntry
{
    [DataMember]
    public String OrderEntryID { get; set; }
    [DataMember]
    public DateTime OrderEntryDate { get; set; }
    [DataMember]
    public Customer OrderCustomer { get; set; }
    [DataMember]
    public List < OrderedProducts > OrderOrderedProducts { get; set; }
}

[DataContract]
public class Customer
{
    [DataMember]
    public string CustomerName { get; set; }
    [DataMember]
    public string CustomerAddressLine1 { get; set; }
    [DataMember]
    public string CustomerAddressLine2 { get; set; }
    [DataMember]
```

```
    public string CustomerCountryCode { get; set; }
}

[DataContract]
public class OrderedProducts
{
    [DataMember]
    public string ProductID { get; set; }
    [DataMember]
    public int Quantity { get; set; }
    [DataMember]
    public string ProductName { get; set; }
}
```

Добавьте к проекту интерфейс `IOrderEntryService` и следующие инструкции `using`.

```
using System.ServiceModel;
using System.Runtime.Serialization;
```

Интерфейс определите следующим образом.

```
[ServiceContract]
[ServiceKnownType(typeof(HQOrderEntryServiceInterface.HQOrderEntry))]
public interface IOrderEntryService
{
    [OperationContract(IsOneWay = true, Action = "*")]
    void SendOrderEntry(System.ServiceModel.
                        MsmqIntegration.MsmqMessage
        <HQOrderEntryServiceInterface.HQOrderEntry>
                        orderEntry);
}
```

Здесь определен метод, который получает сообщение MSMQ, содержащее `OrderEntry`. Кроме того, указано, что это однонаправленная операция, необходимая при работе с очередями. С помощью `ServiceKnownType` указано, что данные в сообщении могут быть сериализированы в класс `OrderEntry`.

Создание `HelperLib`

Перед тем как продолжить работу с решением, надо создать библиотеку с некоторыми обобщенными функциями, которые пригодятся в разных частях проекта. `HelperLib` представляет собой библиотеку классов, используемую во множестве проектов. Она содержит статический класс `GenericSerializer` с методами для сериализации объектов в строки и обратной десериализации в объекты.

Имеются два способа сериализации и десериализации. Первый способ — с применением `XmlSerializer`, который не имеет отношения к WCF. Он необходим для приложения `OrderEntry`. Второй использует часть WCF `DataContractSerializer`. Мы будем использовать оба варианта.

Сначала обратимся из проекта к библиотеке `System.Runtime.Serialization`. Добавим в проект `HelpLib` класс `GenericSerializer.cs` и приведенный ниже код.

Ниже приведены необходимые инструкции `using`.

```
using System.Xml;
using System.Xml.Serialization;
```

```
using System.IO;

using System.Runtime.Serialization;
```

А вот код упомянутых функций.


```
public static class GenericSerializer < T >
{
    public static string Serialize(T p)
    {
        XmlSerializer ser = new XmlSerializer(typeof(T));
        StringWriter sw = new StringWriter();
        ser.Serialize(sw, p);

        return sw.ToString();
    }

    public static T Deserialize(string xml)
    {
        XmlSerializer ser = new XmlSerializer(typeof(T));
        StringReader sr = new StringReader(xml);

        return (T)ser.Deserialize(sr);
    }

    static public string SerializeDC(T o)
    {
        DataContractSerializer dataContractSerializer =
            new DataContractSerializer(typeof(T));
        StringWriter stringWriter = new StringWriter();

        XmlWriter xmlWriter = XmlWriter.Create(stringWriter);

        dataContractSerializer.WriteObject(xmlWriter, o);
        xmlWriter.Close();

        return (stringWriter.ToString());
}

    static public T DeserializeDC(string Xml)
    {
        DataContractSerializer dataContractSerializer =
            new DataContractSerializer(typeof(T));

        StringReader stringReader = new StringReader(Xml);
        XmlReader xmlReader = XmlReader.Create(stringReader);
        T obj = (T)dataContractSerializer.ReadObject(xmlReader);

        return obj;
    }
}
```

Файл CreatingtheCommunicationandIntegrationCase.zip

Создание OrderEntryImplementation

Добавим в папку решения Services библиотеку классов HQOrderEntryImplementation. Эта библиотека содержит реализацию OrderEntryServiceInterface и будет обрабатывать входные сообщения в очереди. Она будет считывать сообщения с использованием привязки msmqIntegrationBinding, которую мы настроим позже, когда создадим HQOrderEntryServiceHost.

Эта библиотека классов будет обращаться к HQOrderEntryServiceInterface, HelperLib, System.Runtimte.Serialization и System.ServiceModel.

Добавим класс, который назовем HQOrderEntryService.cs. Этот класс реализует IOrderEntryService в пространстве имен HQOrderEntryServiceInterface (см. приведенный ниже код).

```
public class HQOrderEntryService :
    HQOrderEntryServiceInterface.IOrderEntryService
{
    public void SendOrderEntry
    (MsmqMessage <HQOrderEntryServiceInterface.HQOrderEntry>
            orderEntryMsg)
    {
        // Код будет добавлен позже
    }
}
```

Создание HQOrderEntryServiceHost

Данное приложение выступает в роли хоста для службы OrderEntryService. Добавьте консольное приложение в папку Service решения и назовите его HQOrderEntryServiceHost. Этот проект будет обращаться к библиотекам HQOrderEntryImplementation и OrderEntryServiceInterface. Кроме того, ему потребуются обращения к System.ServiceModel и System.Runtime.Serialization.

В класс программы добавьте следующие инструкции using.

```
using System.Runtime.Serialization;
using System.ServiceModel;
```

В главном методе добавьте обработчик try catch и несколько инструкций вывода строк на экран для указания состояния хоста службы. Последняя строка представляет собой операцию ReadKey(), гарантирующую, что приложение останется открытым и готовым к обработке сообщений.

```
Console.WriteLine("Started");
try
{
    // Код будет добавлен позже
    Console.WriteLine("OK");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    if (ex.InnerException != null)
    {
```

```
        Console.WriteLine(ex.InnerException.Message);
    }
}
Console.ReadKey();
```

В блоке `try` объявлена переменная типа `ServiceHost`, которая затем инстанцируется с обращением к реализации `HQOrderEntryService`, и, наконец, открывается хост службы.

```
ServiceHost serviceHostOrderEntryService;
serviceHostOrderEntryService = newServiceHost(
    typeof(HQOrderEntryImplementation.HQOrderEntryService));
serviceHostOrderEntryService.Open();
```

Добавьте к проекту конфигурационный файл приложения, а в него добавьте службу `HQOrderEntryImplementation.HQOrderEntryService`.

Эта служба имеет одну конечную точку со следующими свойствами.

- Адрес: `msmq.formatname:DIRECT=OS:.\private$\OrderEntryQueue`
- Привязка: `MsmqIntegrationBinding`
- Контракт: `HQOrderEntryServiceInterface.IOrderEntryService`

Необходимо также более детально указать свойства привязки `msmqIntegrationBinding`. Установите значение `exactlyOnce` равным `false`, так как используется нетранзакционная очередь. Для целей тестирования установите режим безопасности равным `None`, чтобы избежать необходимости активизировать интеграцию с Active Directory. Файл `app.config` должен иметь следующий вид.

```
<?xml version="1.0"?>
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="HQOrderEntryImplementation.HQOrderEntryService">
        <endpoint address=
          "msmq.formatname:DIRECT=OS:.\private$\OrderEntryQueue"
          binding="msmqIntegrationBinding"
          contract=
          "HQOrderEntryServiceInterface.IOrderEntryService"
          bindingConfiguration="BindingMSMQ"/>
      </service>
    </services>

    <bindings>
      <msmqIntegrationBinding>
        <binding name="BindingMSMQ"
                 exactlyOnce="false">
          <security mode="None"/>
        </binding>
      </msmqIntegrationBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Создание OrderEntryApplication

Добавьте в папку Client решения приложение типа Windows forms и назовите его OrderEntryApplication.

Этот проект будет обращаться к библиотекам HelperLib, HQOrderEntryInterface, а также System.Messaging (которая содержит API для отправки сообщения в очередь MSMQ).

Добавьте в форму метод SendMessage с приведенным ниже кодом. Этот метод получает в качестве параметра строку и пересылает эти данные в очередь. Он инстанцирует объект типа MessageQueue для OrderEntryQueue, инстанцирует сообщение и конструирует XMLDocument. Свойство InnerXML объекта XMLDocument устанавливается равным указанным данным, а свойство Body сообщения — этому XMLDocument. После этого messageQueue пересылает сообщение.

```
private static void SendMessage(string data)
{
    MessageQueue messageQueue;
    messageQueue =
        new MessageQueue(@".\Private$\OrderEntryQueue");

    System.Messaging.Message message;
    message = new System.Messaging.Message();

    System.Xml.XmlDocument xmlDocument;
    xmlDocument = new XmlDocument();

    xmlDocument.InnerXml = data;

    message.Body = xmlDocument;
    messageQueue.Send(message);
}
```

Данный метод требует получения данных в формате XML, который содержит сериализированное сообщение. Для создания этого XML добавьте функцию GetXML-ForOrderEntry, которая имеет следующий вид.

```
private static string GetXMLForOrderEntry ()
{
    string tmp;
    HQOrderEntryServiceInterface.HQOrderEntry test;
    test = new HQOrderEntryServiceInterface.HQOrderEntry();

    test.OrderEntryID = "00000001";
    test.OrderEntryDate = System.DateTime.Now;
    test.OrderCustomer =
        new HQOrderEntryServiceInterface.Customer();
    test.OrderCustomer.CustomerName = "WROX";
    test.OrderCustomer.CustomerAddressLine1 =
        "CustomerAddressLine1";
    test.OrderCustomer.CustomerAddressLine2 =
        "CustomerAddressLine2";
    test.OrderCustomer.CustomerCountryCode = "BE";
    test.OrderOrderedProducts =
        new List
```

```
            <HQOrderEntryServiceInterface.OrderedProducts>();
    test.OrderOrderedProducts.Add(
        new HQOrderEntryServiceInterface.OrderedProducts());
    test.OrderOrderedProducts[0].ProductID = "P08872";
    test.OrderOrderedProducts[0].Quantity = 5;
    test.OrderOrderedProducts[0].ProductName = "Car";
    test.OrderOrderedProducts.Add(
    new HQOrderEntryServiceInterface.OrderedProducts()
    );
    test.OrderOrderedProducts[1].ProductID = "P02287";
    test.OrderOrderedProducts[1].ProductName = "Bike";
    test.OrderOrderedProducts[1].Quantity = 5;
    tmp = HelperLib.GenericSerializer
        <HQOrderEntryServiceInterface.HQOrderEntry>.
        Serialize(test);
    return tmp;
}
```

Эта функция инстанцирует класс `OrderEntry`, устанавливает осмысленные значения его членов, а затем использует функцию `Serialize` из `HelperLib` для получения XML.

Теперь в форму можно добавить кнопку, которая будет выполнять код.

```
SendMessage(GetXMLForOrderEntry());
```

Можно поэкспериментировать с другими кнопками, отправляющими в очередь сообщения с другим содержимым.

Создание `LocalOrderEntryInterface`

Добавим проект библиотеки классов `LocalOrderEntryInterface` и интерфейс `IReceiveOrderEntryLocalBranch`. Этот интерфейс содержит контракт службы для локальных служб `OrderEntry` в различных филиалах.

Начнем с добавления ссылок на библиотеки `System.ServiceModel` и `System.Runtime.Serialization`.

Добавьте в проект интерфейс `IReceiveOrderEntryLocalBranch.cs`. Внесите в этот файл необходимые инструкции `using`.

```
using System.Runtime.Serialization;
using System.ServiceModel;
```

Добавьте следующий код.


```
[ServiceContract]
public interface IReceiveOrderEntryLocalBranch
{
    [OperationContract]
    int SendLocalOrderEntry(LocalOrderEntry localOrderEntry);
}

[DataContract]
public class LocalOrderEntry
{
    [DataMember]
    public String OrderEntryID { get; set; }
```

```
    [DataMember]
    public DateTime OrderEntryDate { get; set; }
    [DataMember]
    public string CustomerName { get; set; }
    [DataMember]
    public string CustomerAddressLine1 { get; set; }
    [DataMember]
    public string CustomerAddressLine2 { get; set; }
    [DataMember]
    public string CustomerCountryCode { get; set; }
    [DataMember]
    public List<OrderedProducts>
        OrderOrderedProducts { get; set; }
}

[DataContract]
public class OrderedProducts
{
    [DataMember]
    public string ProductID { get; set; }
    [DataMember]
    public int Quantity { get; set; }
    [DataMember]
    public string LocalizedDescription { get; set; }
}
```

Файл CreatingtheCommunicationandIntegrationCase.zip

Продолжение разработки метода `OrderEntryImplementation`

Теперь надо реализовать поток процесса получения заказа методом `SendOrderEntry`. При получении сообщение проверяется на корректность содержащегося в нем заказа. Если он корректен, то сообщение преобразуется в структуру `localOrderEntry`. Такое преобразование предпринимает действия по переводу описания товара. Если же заказ некорректен, он отправляется обратно в очередь.

Библиотека `OrderEntryService` требует обращения к библиотекам `HelperLib`, `LocalOrderEntryInterface`, `OrderEntryInterface`, `System.ServiceModel`, `System.Runtime.Serialization` и `System.Transactions`.

Добавьте в файл `HQOrderEntryService.cs` проекта `OrderEntryImplementation` приведенные далее "заглушки" методов. Позже мы заполним их необходимым кодом. Нам нужны методы для проверки корректности заказа, преобразования схемы, маршрутизации записи заказа и отправки заказа в очередь некорректных заказов.

```
private bool CheckIfOrderIsValid
    (HQOrderEntryServiceInterface.HQOrderEntryorderEntry)
{
    // Необходимый код будет добавлен позже
    return true;
}

private LocalOrderEntryInterface.LocalOrderEntry
    ConvertOrderEntrySchema(
      HQOrderEntryServiceInterface.HQOrderEntry orderEntry)
```

```
{
    // Необходимый код будет добавлен позже
    return null;
}

private void RouteOrderEntry
    (LocalOrderEntryInterface.LocalOrderEntry localOrderEntry)
{
    // Необходимый код будет добавлен позже
}

private void SendToInvalidOrderQueue
    (MsmqMessage <HQOrderEntryServiceInterface.HQOrderEntry>
        orderEntryMsg)
{
    // Необходимый код будет добавлен позже
}

private string TranslateProductDescription(string productID,
    string languageCode)
{
    // Необходимый код будет добавлен позже
    return "";
}
```

После создания "заглушек" можно записать поток процесса в методе `SendOrderEntry`.

```
public void SendOrderEntry(
MsmqMessage <HQOrderEntryServiceInterface.HQOrderEntry>
             orderEntryMsg)
{
    try
    {
        if (CheckIfOrderIsValid(orderEntryMsg.Body))
        {
            RouteOrderEntry(ConvertOrderEntrySchema(
                    orderEntryMsg.Body));
        }
        else
        {
            SendToInvalidOrderQueue(orderEntryMsg);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Создание службы HQProductServiceASMX

Чтобы создать службу `HQProductServiceASMX`, надо выполнить несколько действий. Сначала надо создать веб-службу, а затем добавить ее в реализацию по ссылке. И наконец, надо создать методы `CheckIfOrderIsValid`, `TranslateProductDescription` и `ConvertOrderEntrySchema`.

Создание веб-службы

Заказ корректен, если все заказанные товары могут быть доставлены в страну заказчика. Эта проверка выполняется с помощью веб-службы ASMX на основе значения идентификатора `productId` и кода `countryCode` заказчика.

Добавьте новый веб-сайт в папку `Services` решения. Для этого щелкните правой кнопкой мыши на папке решения и выберите из контекстного меню пункт Add⇨New Website. В диалоговом окне Add New Web Site выберите ASP.NET Web Service (рис. 12.4).

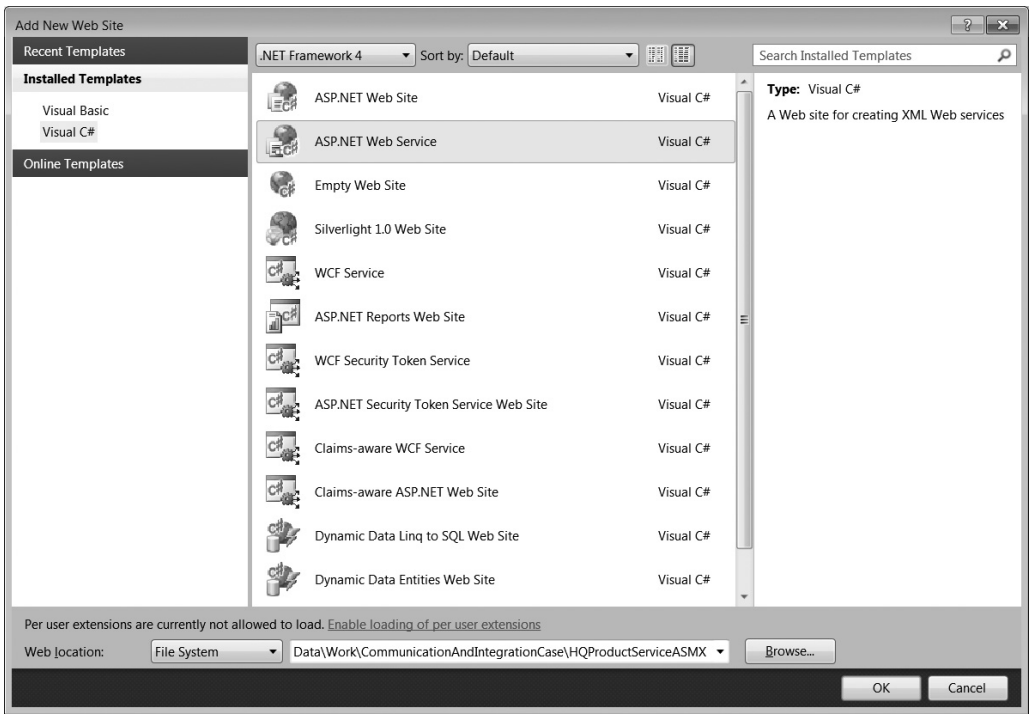


Рис. 12.4. Добавление нового веб-сайта

Переименуйте файл `Service.asmx` в `ProductService.asmx` и откройте файл `Service.cs` в каталоге `App_Code`. В этом файле удалите метод `HelloWorld` и добавьте код метода `IsProductAvailableForCountry`. Этот метод получает `productID` и `countryCode` в качестве параметров и возвращает логическое значение

```
[WebMethod]
public bool IsProductAvailableForCountry(string productID,
                                         string countryCode)
{
}
```

Для простоты реализации воспользуемся жестко закодированным алгоритмом, который вместо проверки доступности продукта по базе данных возвращает значение `true` или `false` в зависимости от кода страны. Будем считать товар доступным для

стран с countryCode равным BE, AR или AT; в противном случае считаем его недоступным.

```
[WebMethod]
public bool IsProductAvailableForCountry(string productID,
                                         string countryCode)
{
    if ((countryCode == "BE") ||
        (countryCode == "AR") ||
        (countryCode == "AT"))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Необходимо настроить данный веб-сайт таким образом, чтобы он использовал статический номер порта, а не динамические порты. Выберите порт номер 8081 и откройте окно свойств веб-сайта, нажав клавишу <F4>. Установите значение Use Dynamic Ports равным False, а значение Port Number – равным 8081 (рис. 12.5).

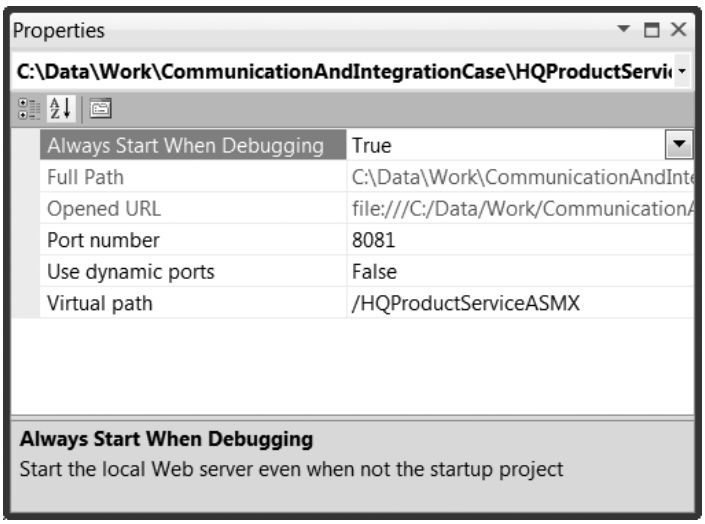


Рис. 12.5. Окно свойств веб-сайта

Добавление HQProductServiceASMX к OrderEntryServiceImplementation

Добавьте ссылку на службу в проект HQOrderEntryImplementation.

Щелкните правой кнопкой мыши на ссылках и выберите Add Service Reference. В открывшемся диалоговом окне Add Service Reference щелкните на кнопке Discover; при этом вы должны увидеть операцию в HQProductService (рис. 12.6).

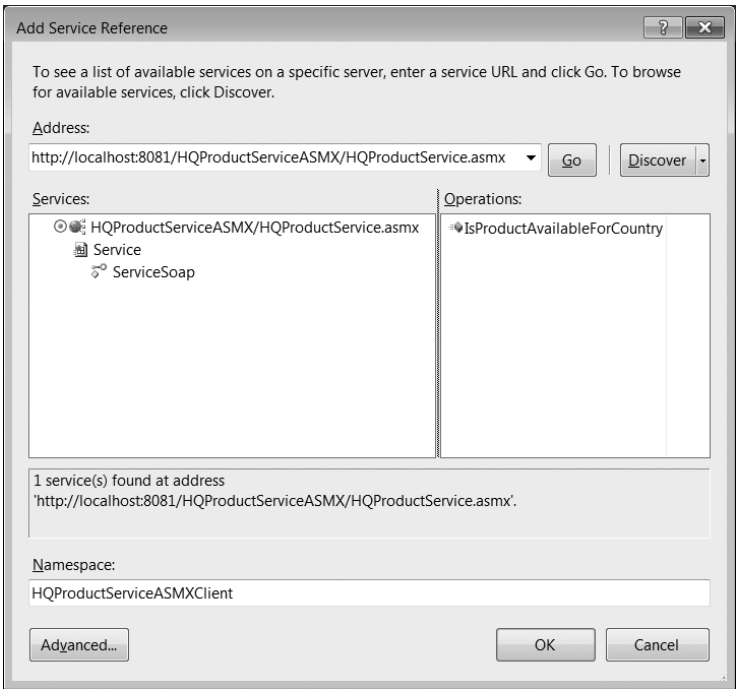


Рис. 12.6. Диалоговое окно Add Service Reference

Установите пространство имен `HQProductServiceASMXClient` и щелкните на кнопке OK.

В проекте `HQOrderEntryImplementation` будет сгенерирован файл `app.config`, который можно удалить, так как нам надо добавить конфигурацию к `HQOrderEntryServiceHost`.

Кодирование метода `CheckIfOrderIsValid`

Теперь служба `HQProductService` добавлена в качестве ссылки, так что метод `CheckIfOrderIsValid` можно разработать, например, так, как показано ниже.

```
private bool CheckIfOrderIsValid
    (HQOrderEntryServiceInterface.HQOrderEntry orderEntry)
{
    HQProductServiceASMXClient.ServiceSoapClient client;
    client =
        new HQProductServiceASMXClient.ServiceSoapClient();
    bool orderIsValid;
    orderIsValid = true;
    foreach (var item in orderEntry.OrderOrderedProducts)
    {
        orderIsValid = client.IsProductAvailableForCountry(
        item.ProductID,
        orderEntry.OrderCustomer.CustomerCountryCode);
    }
    return orderIsValid;
}
```

Кодирование метода TranslateProductDescription

Для обращения к службе TranslateProductDescription с применением REST не требуется добавлять ссылку на службу или создавать прокси. Надо воспользоваться HttpWebRequest и указать languageCode и productID как часть URL для обращения к службе. Служба отвечает строкой, сериализированной в виде контракта данных WCF, содержащей перевод. Далее ее надо десериализировать с помощью метода DeserializeDC из HelperLib.

```
private string TranslateProductDescription(string productID,
                                           string languageCode)
{
    System.Net.HttpWebRequest webrequest;
    webrequest =
        (System.Net.HttpWebRequest)
        System.Net.HttpWebRequest.Create(
        string.Format
        (@"http://localhost:8081/HQLocalizationService/"
        "TranslateProductDescriptions/{0}/{1}",
        languageCode,
        productID));
    webrequest.ContentLength = 0;
    System.Net.HttpWebResponse webresponse;
    webresponse =
        (System.Net.HttpWebResponse)webrequest.GetResponse();
    Encoding enc = System.Text.Encoding.GetEncoding(1252);
    System.IO.StreamReader loResponseStream = new
    System.IO.StreamReader(webresponse.GetResponseStream(),
                           enc);
    string response = loResponseStream.ReadToEnd();
    string answer;
    answer = GenericSerializer<string>.
             DeserializeDC(response);
    return answer;
}
```

Кодирование метода ConvertOrderEntrySchema

Этот метод получает в качестве входных данных HQOrderEntry и возвращает результат LocalOrderEntry. Метод создает новый объект LocalOrderEntry и заполняет его данными из HQOrderEntry. Он циклически проходит по всем товарам в заказе, вызывая для каждого из них TranslateProductDescription, и добавляет их (с префиксом страны) в локальный заказ.

Чтобы обратиться к классу LocalOrderEntry, сначала надо добавить ссылку на проект LocalOrderEntryInterface.

Доступно для загрузки на Wrox.com

```
private LocalOrderEntryInterface.LocalOrderEntry
    ConvertOrderEntrySchema(
        HQOrderEntryServiceInterface.HQOrderEntry orderEntry)
{
    LocalOrderEntryInterface.LocalOrderEntry localOrderEntry;
    localOrderEntry =
        new LocalOrderEntryInterface.LocalOrderEntry();
```

```
    localOrderEntry.CustomerName =
        orderEntry.OrderCustomer.CustomerName;
    localOrderEntry.CustomerAddressLine1 =
        orderEntry.OrderCustomer.CustomerAddressLine1;
    localOrderEntry.CustomerAddressLine2 =
        orderEntry.OrderCustomer.CustomerAddressLine2;
    localOrderEntry.CustomerCountryCode =
        orderEntry.OrderCustomer.CustomerCountryCode;

    localOrderEntry.OrderOrderedProducts =
        new List<LocalOrderEntryInterface.OrderedProducts>();

    foreach (var item in orderEntry.OrderOrderedProducts)
    {
        string translation;
        translation = TranslateProductDescription(
            item.ProductID,
            orderEntry.OrderCustomer.CustomerCountryCode);

        localOrderEntry.OrderOrderedProducts.Add(
            new LocalOrderEntryInterface.OrderedProducts
            { ProductID =
                orderEntry.OrderCustomer.CustomerCountryCode
                + "/"
                + item.ProductID, Quantity = item.Quantity,
                LocalizedDescription = translation });
    }
    return localOrderEntry;
}
```

Файл CreatingtheCommunicationandIntegrationCase.zip

Создание HQLocalizationService

Это служба REST, предоставляемая WCF и размещенная в консольном приложении. Это означает, что в качестве хоста не применяются ни стандартный WCF ServiceHost, ни IIS, а вместо них используется `WebServiceHost`. Мы не будем разделять интерфейсы, реализацию и хост на три проекта. Применение подхода REST избавляет от необходимости иметь интерфейсы, используемые приложением, отличным от самой службы.

Добавьте консольное приложение в папку `Services` решения и внесите в него ссылки на `System.Runtime.Serialization`, `System.ServiceModel` и `System.ServiceModel.Web`.

Добавьте интерфейс `ITranslateProductDescriptions.cs`. В этот файл должна быть включена инструкция `using` для пространства имен `System.ServiceModel.Web`, а также код, приведенный ниже.

```
[ServiceContract]
public interface ITranslateProductDescriptions
{
    [OperationContract]
    [WebGet(UriTemplate = @"/{languageCode}/{productID}")]
    string GetProductDescription(string productID,
                                 string languageCode);
}
```

Здесь определяется метод GetProductDescription, который получает два параметра для возврата соответствующего перевода. Перед методом следует поместить атрибут WebGet, указывающий шаблон URI. Этот шаблон отображает данные из URL на входные параметры метода.

Добавьте класс для реализации этого интерфейса. Пока что этот код не выполняет реальный перевод, а просто возвращает жестко закодированную строку.

```
public class TranslateProductDescriptions
          :ITranslateProductDescriptions
{
    public string GetProductDescription(string productID,
                                        string languageCode)
    {
        return "Translated";
    }
}
```

Для запуска хоста и открытия URL необходимо добавить следующий код в консольное приложение.

```
Console.WriteLine("TranslateProductDescriptions");
WebServiceHost webServiceHost;
webServiceHost =
    new WebServiceHost(typeof(TranslateProductDescriptions));
webServiceHost.Open();
Console.ReadKey();
```

Необходимо также добавить инструкцию using для System.ServiceModel.Web. Конфигурирование хоста выполняется следующим образом.

```
<?xml version="1.0"?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="TranslateProductDescriptions">
        <endpoint
          address=
            "http://localhost:8082/HQLocalizationService/
↳               TranslateProductDescriptions"
            binding="webHttpBinding"
            contract="ITranslateProductDescriptions" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

В качестве привязки данной службы используется webHttpBinding.

Кодирование метода RouterOrderEntry

Теперь, когда у нас есть метод для преобразования заказа в локальный заказ, можно приступить к разработке метода для маршрутизации заказа далее, в пункт назначения. Для этого надо сначала создать прокси для интерфейса IReceiveOrderEntryLocalBranch. Добавьте к проекту HQOrderEntryImplementation файл LocalOrderEntryProxy.cs и все необходимые инструкции для включения System.ServiceModel. Вставьте код, приведенный ниже.

```
public class LocalOrderEntryProxy :
ClientBase
  <LocalOrderEntryInterface.IReceiveOrderEntryLocalBranch>,
LocalOrderEntryInterface.IReceiveOrderEntryLocalBranch
{
    public LocalOrderEntryProxy()
    : base("LocalOrderEntryEndpoint")
    {
    }
    public int SendLocalOrderEntry(
      LocalOrderEntryInterface.LocalOrderEntry
        localOrderEntry)
    {
        return Channel.SendLocalOrderEntry(localOrderEntry);
    }
}
```

Прокси представляет собой класс, унаследованный от `ClientBase<T>` и реализующий интерфейс `IReceiveOrderEntryLocalBranch`. Этот интерфейс также представляет собой обобщенный тип для базового класса. Конструктор класса получает строку, которая является именем конечной точки в конфигурации. Метод `SendLocalOrderEntry` пересылает данные `localOrderEntry` в канал путем вызова метода свойства канала `ClientBase` с тем же именем.

При наличии этого прокси можно изменить метод `RouteOrderEntry` так, как показано ниже.

```
private void RouteOrderEntry
  (LocalOrderEntryInterface.LocalOrderEntry localOrderEntry)
{
    try
    {
        LocalOrderEntryProxy localOrderEntryProxy;
        localOrderEntryProxy = new LocalOrderEntryProxy();
        int a;
        a = localOrderEntryProxy.SendLocalOrderEntry(
                localOrderEntry);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Создание RealTimeOrderTrackingApplication

`RealTimeOrderTrackingApplication` представляет собой приложение, которое получает сигнал для каждого обработанного службой `HQOrderEntry` заказа и выводит количество заказов для каждой страны. Для реализации такого приложения воспользуемся моделью “издатель–подписчик”. Приложение подписывает себя на публикации, поступающие от `HQOrderEntryService`. Подписка выполняется путем вызова операции, предоставляемой службой подписки, размещенной на хосте `HQServiceHost`. При вызове этой операции служба запоминает вызывающего, сохраняя ссылку на канал обратного вызова в списке в памяти службы. Когда заказ размещен, служба вызывает

метод этого канала, который является частью интерфейса обратного вызова, известного службе и реализованного вызывающей стороной. Это означает, что в момент публикации служба OrderEntry действует как клиент и пересылает данные службе, реализованной в отслеживающем приложении.

Сначала определим контракт обратного вызова, который располагается в отдельном проекте.

Добавьте новую библиотеку классов RealTimeOrderTrackingCallBackContract в папку ServiceIntefaces решения. Добавьте библиотеку System.ServiceModel как ссылку и интерфейс IOrderTracking с кодом

```
[ServiceContract]
public interface IOrderTracking
{
    [OperationContract]
    void NewOrderForCountry(string countryID);
}
```

Кодирование метода RealTimeOrderTrackingApplication

Рассматриваемое приложение является приложением Windows. Добавьте это приложение, именуемое RealTimeOrderTrackingApplication, в каталог Clients решения. Приложение требует обращения к System.ServiceModel и RealTimeOrderTrackingCallBackContract.

Реализация RealTimeOrderTrackingCallBackContract создается путем добавления класса CallBackImplementation. Создайте класс наподобие следующего.

```
public class CallBackImplementation :
    RealTimeOrderTrackingCallBackContract.IOrderTracking
{
    Dictionary <string,int> NumberOfOrderEntries;
    public CallBackImplementation()
    {
        NumberOfOrderEntries = new Dictionary<string,int>();
    }
    public void NewOrderForCountry(string countryID)
    {
    }
}
```

Конструктор этого класса инстанцирует словарь строк и чисел типа int. Он необходим для хранения количества заказов для каждой страны. В этом обобщенном списке строка представляет код страны, а int — количество заказов.

В методе NewOrderForCountry можно выводить количество заказов в виде списка или диаграммы. Поскольку эта проблема связана с графическим пользовательским интерфейсом, а не с WCF, код этой функции в книгу не включен.

Добавление интерфейса ISubscribeToOrderTrackingInfo

Для подписки на отслеживание количества заказов требуется интерфейс с методами для подписки и отписки, который размещается в проекте HQOrderEntryServiceInterface. Он должен быть осведомлен о CallbackContract, так что начнем с добавления ссылки на RealTimeOrderTrackingCallBackContract в проект HQOrderEntryServiceInterface.

Добавьте интерфейс `ISubscribeToOrderTrackingInfo` и необходимые пространства имен.

```
using System.ServiceModel;
using System.Runtime.Serialization;
```

Интерфейс имеет следующий вид.

```
[ServiceContract(CallbackContract =
typeof(RealTimeOrderTrackingCallBackContract.IOrderTracking))]
    public interface ISubscribeToOrderTrackingInfo
{
    [OperationContract]
    void Subscribe();
    [OperationContract]
    void UnSubscribe();
}
```

Этот интерфейс похож на контракт обычной службы и использует параметр `CallBackContract` атрибута `ServiceContract` для ссылки на контракт обратного вызова.

Реализация метода `SubscribeService`

Реализация этой службы располагается в проекте `HQOrderEntryServiceImplementation`. Перейдите к этому проекту и добавьте в него ссылку на библиотеку `RealTimeOrderTrackingCallBackContract`. Затем добавьте класс `SubscribeService`, который будет реализовывать интерфейс `ISubscribeToOrderTrackingInfo`. Данная реализация приведена ниже.


```
[ServiceBehavior(InstanceContextMode =
                InstanceContextMode.Single)]
public class SubscribeService :
    OrderEntryServiceInterface.ISubscribeToOrderTrackingInfo
{
    List <RealTimeOrderTrackingCallBackContract.IOrderTracking>
            callBacks;
    public SubscribeService()
    {
        callBacks =
            new List<RealTimeOrderTrackingCallBackContract.
                    IOrderTracking> ();
    }

    public void Subscribe()
    {
        Console.WriteLine("**Someone Subscribed");
        callBacks.Add(
            System.ServiceModel.OperationContext.Current
                  .GetCallbackChannel
            <RealTimeOrderTrackingCallBackContract.
             IOrderTracking>());
    }

    public void PublishOrderEntrySignal(string countryID)
```

```
    {
        foreach (var item in callBacks)
        {
            item.NewOrderForCountry(countryID);
        }
    }

    public void UnSubscribe()
    {
        Console.WriteLine("**Someone UnSubscribed");
        callBacks.Remove(
            System.ServiceModel.OperationContext.Current.
                   GetCallbackChannel
               <RealTimeOrderTrackingCallBackContract.
                  IOrderTracking>());
    }
}
```

Файл CreatingtheCommunicationandIntegrationCase.zip

Здесь видно, что служба работает в режиме `InstanceContextMode.Single`. Таким образом, выполняется единственное инстанцирование службы, и этот экземпляр создается до первого вызова и будет сохраняться в памяти и после него. Это необходимо, поскольку требуется запоминание списка каналов обратного вызова.

Список подписчиков представляет собой список ссылок на интерфейсы обратного вызова (`IOrderTracking`). Когда WCF вызывает метод подписки, доступ к каналу обратного вызова получается с использованием класса `OperationContext`. Это — статический класс, позволяющий вам получить канал обратного вызова. Тип этого канала, добавляемого в список, — `IOrderTracking`. Отписка выполняется простым удалением его из списка.

Имеется также функция `PublishOrderEntrySignal`, которая принимает идентификатор страны в качестве параметра и проходит по всем подписанным каналам обратного вызова, вызывая для них метод `NewOrderForCountry`.

Вызов подписчиков при обработке заказа

При обработке заказа необходимо вызвать все каналы обратного вызова. Это можно сделать путем вызова `PublishOrderEntrySignal` из `SubscribeService`. Таким образом, нам нужен доступ к `SubscribeService` из реализации операции `SendOrderEntry`. Для этого надо воспользоваться другим классом со статической ссылкой на `SubscribeService`. Добавьте класс `SubscriberSingleton` с кодом, приведенным ниже.

```
public class SubscriberServiceSingleton
{
    private static SubscribeService instance;
    private static readonly object singletonLock =
        new object();

    public static SubscribeService GetInstance()
    {
        if (instance == null)
        {
            lock (singletonLock)
            {
```

```
                if (instance == null)
                {
                    instance = new SubscribeService();
                }
            }
        }
        return instance;
    }
}
```

Данный класс имеет метод для получения экземпляра `SubscribeService`. При вызове данного метода он при необходимости инстанцирует `SubscribeService` и возвращает указанный экземпляр.

Поскольку метод `GetInstance` открытый и статический, он может быть вызван службой `HQOrderEntry`. Теперь можно добавить вызов для опубликования сигнала в реализации `SendOrderEntry`.

```
if (CheckIfOrderIsValid(orderEntryMsg.Body))
{
    Console.WriteLine("Order Is VALID");
    RouteOrderEntry(ConvertOrderEntrySchema(
                    orderEntryMsg.Body));
    HQOrderEntryImplementation.SubscriberServiceSingleton
        .GetInstance().PublishOrderEntrySignal(
    orderEntryMsg.Body.OrderCustomer.CustomerCountryCode);
}
else
{
    Console.WriteLine("Order Is not VALID");
    SendToInvalidOrderQueue(orderEntryMsg);
}
```

Открытие SubscribeService

Данную службу надо открыть в главном методе `HQOrderEntryServiceHost`. В этом коде инстанцируется `SubscribeService` с помощью класса `SubscribeService-Singleton`. Ссылка на службу передается хосту службы во время конструирования. В результате `SubscribeService` доступен всем службам, размещенным в данном приложении.

```
ServiceHost serviceHostSubscribeService;
OrderEntryServiceImplementation.SubscribeService
    subscribeService;
subscribeService =
    HQOrderEntryImplementation.SubscriberServiceSingleton.
        GetInstance();
serviceHostSubscribeService = new ServiceHost(subscribeService);
serviceHostSubscribeService.Open();
```

Подписка из RealTimeOrderTrackingApplication

Этому приложению требуется ссылка на `HQOrderEntryServiceInterface`.

Добавьте в форму кнопку, которая запускает подписку. Код, запускаемый кнопкой, инстанцирует класс `CallBackImplementation` и "заворачивает" его в `InstanceContext`.

```
CallBackImplementation callBack;
callBack = new CallBackImplementation();
InstanceContext instanceContext =
    new InstanceContext(callBack);
```

Затем используем `ChannelFactory` из интерфейса `ISubscribeToOrderTrackingInfo`. Это делается с помощью класса `DuplexChannelFactory`, который получает "обертку" `instanceContext` вокруг `CallBackImplementation` в качестве параметра вместе с привязкой `NetTcpBinding`.

```
ChannelFactory
    <OrderEntryServiceInterface.ISubscribeToOrderTrackingInfo>
    cf = new DuplexChannelFactory
    <OrderEntryServiceInterface.ISubscribeToOrderTrackingInfo>
    (instanceContext, new System.ServiceModel.NetTcpBinding());

HQOrderEntryServiceInterface.ISubscribeToOrderTrackingInfo
    subscriber = cf.CreateChannel(new
        EndpointAddress("net.tcp://localhost:9875"));

subscriber.Subscribe();
```

После того как канал создан с применением корректного адреса, можно вызывать метод подписки.

Настройка `HQOrderEntryServiceHost`

Приложение `HQOrderEntryServiceHost` размещает две службы и имеет одну клиентскую конечную точку. Полностью вид конфигурации показан ниже.


```
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="HQOrderEntryImplementation.HQOrderEntryService">
        <endpoint
          address="msmq.formatname:DIRECT=
↳                 OS:.\private$\OrderEntryQueue"
          binding="msmqIntegrationBinding"
          contract="HQOrderEntryServiceInterface.
↳                 IOrderEntryService"
          bindingConfiguration="BindingMSMQ"/>
      </service>

      <service
        name="HQOrderEntryImplementation.SubscribeService">
        <endpoint
          address="net.tcp://localhost:9875"
          binding="netTcpBinding"
          contract="HQOrderEntryServiceInterface.
↳                 ISubscribeToOrderTrackingInfo"/>
      </service>
    </services>

    <bindings>
```

```
      <msmqIntegrationBinding>
        <binding name="BindingMSMQ" exactlyOnce="false">
          <security mode="None"/>
        </binding>
      </msmqIntegrationBinding>
    </bindings>

    <client>
      <endpoint
        address="http://localhost:8081/HQProductServiceASMX
↳         /HQProductService.asmx" binding="basicHttpBinding"
         contract="HQProductServiceASMXClient.ServiceSoap"
         name="ServiceSoap"/>
    </client>
  </system.serviceModel>
</configuration>
```

Файл CreatingtheCommunicationandIntegrationCase.zip

Создание маршрутизатора

Для маршрутизатора требуется другое приложение для размещения службы. Этот хост будет реализовывать не функциональный интерфейс, а технический интерфейс, являющийся частью пространства имен WCF `ServiceModel.Routing`. Вместо выполнения кода маршрутизатор будет пересылать входящие сообщения другой службе на основе фильтров, определенных в конфигурации.

Добавьте консольное приложение `RouterHost` в папку `Services` решения и предоставьте ему ссылки на `System.ServiceModel` и `System.ServiceModel.Routing`.

Код основного метода для открытия маршрутизатора приведен ниже.

```
try
{
    Console.WriteLine("RoutingService");
    ServiceHost serviceHost;
    serviceHost = new ServiceHost
        (typeof(System.ServiceModel.Routing.RoutingService));
    serviceHost.Open();
    Console.WriteLine("Started");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Вместо предоставления типа реализации мы предоставляем тип `System.ServiceModel.Routing.RoutingService`.

Весь секрет — в конфигурации. Сначала добавьте службу.

```
<services>
  <service
    behaviorConfiguration="RoutingServiceBehavior"
    name="System.ServiceModel.Routing.RoutingService">
    <endpoint
      address="http://localhost:9874/Router"
      binding="wsHttpBinding"
```

```
      name="RoutingServiceEndpoint"
      contract=
        "System.ServiceModel.Routing.IRequestReplyRouter" />
  </service>
</services>
```

Здесь мы указываем, что вызов размещаемой службы осуществляется `System.ServiceModel.Routing.RoutingService` с контрактом `System.ServiceModel.Routing.IRequestReplyRouter`. Указывается также адрес и привязка.

Данная служба имеет атрибут `behaviorConfiguration` со значением `RoutingServiceBehavior`. Под этим именем имеется элемент `routing`, ссылающийся на `filterTableName`, с именем `routingRules`.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="RoutingServiceBehavior">
      <serviceDebug includeExceptionDetailInFaults="true" />
      <routing filterTableName="routingRules"
               routeOnHeadersOnly="false" />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Этот атрибут `filterTableName` указывает на другой раздел конфигурации, в котором определены два фильтра. Оба они определены с помощью XPath. Первый отфильтровывает сообщения, у которых код страны — `AR`. Второй отфильтровывает сообщения с кодом страны `BE`. Имена обоих фильтров ссылается на `routingRules`, где эти два имени фильтров отображаются на конечные точки.

```
<routing>
  <filters>
    <filter name="Filter_CustomerCountryCode_AR"
            filterType="XPath"
            filterData=
            "boolean(//*[local-name()=
↳             'CustomerCountryCode']/text() = 'AR')"
    />

    <filter name="Filter_CustomerCountryCode_BE"
            filterType="XPath"
            filterData=
            "boolean(//*[local-name()=
↳             'CustomerCountryCode']/text() = 'BE')"
    />
  </filters>

  <filterTables>
    <filterTable name="routingRules">
      <add filterName="Filter_CustomerCountryCode_AR"
           endpointName="ArgentinaEndpoint"/>
      <add filterName="Filter_CustomerCountryCode_BE"
           endpointName="BelgiumEndpoint"/>
    </filterTable>
  </filterTables>
</routing>
```

Эти конечные точки представляют собой обычные конечные точки со своими адресами, привязками и контрактами.

```
<client>
  <endpoint
    binding="wsHttpBinding"
    bindingConfiguration=""
    contract="*"
    address="http://localhost:9874/ArgentinaHost"
    name="ArgentinaEndpoint" kind=""
    endpointConfiguration="">
  </endpoint>

  <endpoint
    binding="netTcpBinding"
    bindingConfiguration=""
    contract="*"
    address="net.tcp://localhost:9871/BelgiumBranch"
    name="BelgiumEndpoint" kind=""
    endpointConfiguration="">
  </endpoint>
</client>
```

Конфигурирование HQOrderEntryServiceHost

Чтобы заставить работать все решение, необходимо корректно сконфигурировать HQOrderEntryServiceHost. Эта конфигурация включает две службы: одну для получения записей заказов из очереди, другую — для подписки. Имеются также два клиента: один для службы ASMX для проверки доступности заказа в стране, второй — для работы с маршрутизатором.

Ниже показано, как выглядит полная конфигурация.


```
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="HQOrderEntryImplementation.HQOrderEntryService">
        <endpoint address="msmq.formatname:DIRECT=
            OS:.\private$\OrderEntryQueue"
          binding="msmqIntegrationBinding"
          contract="HQOrderEntryServiceInterface.
            IOrderEntryService"
          bindingConfiguration="BindingMSMQ"/>
      </service>

      <service
        name="HQOrderEntryImplementation.SubscribeService">
        <endpoint
          address="net.tcp://localhost:9875"
          binding="netTcpBinding"
          contract="HQOrderEntryServiceInterface.
            ISubscribeToOrderTrackingInfo"/>
```

```
      </service>
    </services>

    <bindings>
      <msmqIntegrationBinding>
        <binding name="BindingMSMQ" exactlyOnce="false">
          <security mode="None"/>
        </binding>
      </msmqIntegrationBinding>
    </bindings>

    <client>
      <endpoint
        address="http://localhost:8081/HQProductServiceASMX
↳         /HQProductService.asmx" binding="basicHttpBinding"
        contract="HQProductServiceASMXClient.ServiceSoap"
        name="ServiceSoap"/>
      <endpoint name="LocalOrderEntryEndpoint"
        address="http://localhost:9874/Router"
        binding="wsHttpBinding"
        contract="LocalOrderEntryInterface.
↳         IReceiveOrderEntryLocalBranch"/>
    </client>
  </system.serviceModel>
</configuration>
```

Файл CreatingtheCommunicationandIntegrationCase.zip

13

Создание бизнес-процесса

В ЭТОЙ ГЛАВЕ...

- Поддержка длительных бизнес-процессов со стороны WCF
- Создание декларативного рабочего процесса в Visual Studio
- Реализация контрактов служб в XAML и вызов внешних служб во время рабочего процесса
- Настройка `SQLWorkflowInstanceStore`
- Работа с отправкой и получением и связь входящих и исходящих сообщений

Эта глава представляет собой интерактивное пошаговое руководство по созданию бизнес-процессов в Visual Studio 2010 с использованием WCF и служб рабочих процессов. Она начинается с определения требований к задаче и показывает пошаговую разработку решения. До конца главы мы разработаем бизнес-процесс, хост для его размещения, клиенты и другие службы, которые взаимодействуют с данным процессом. Это завершенный пример, работоспособность которого вы сможете проверить в действии.

Определение требований

Возьмем для нашего примера простой процесс (рис. 13.1), который должен помочь уяснить основные концепции и научить разрабатывать рабочие процессы. Это классический пример бизнес-процесса, имеющего дело с запросами о предоставлении отпусков.

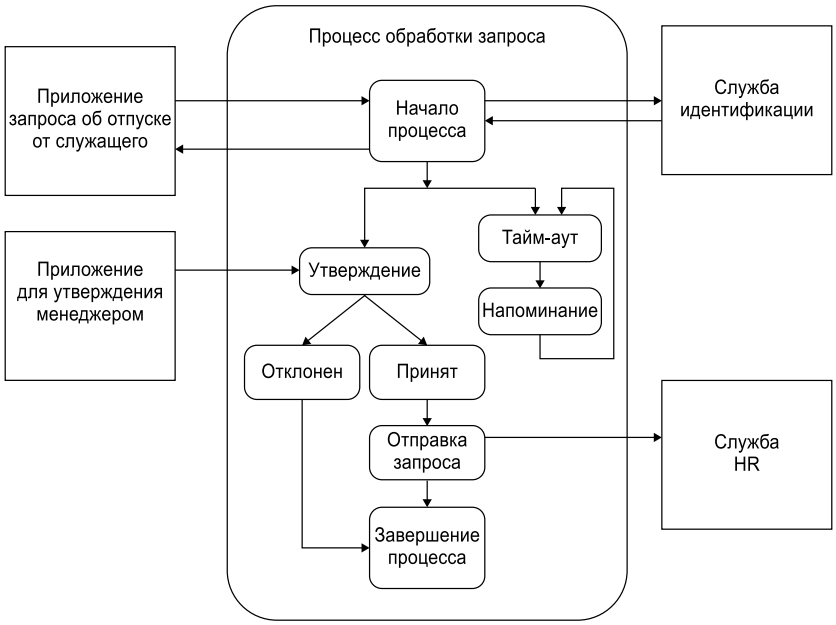


Рис. 13.1. Процесс обработки заявления об отпуске

Компания хочет предоставить рабочий процесс в виде службы, которая получает запрос об отпуске от своих сотрудников, ждет одобрения менеджера и отправляет запрос службе отдела кадров.

Этот рабочий процесс запускается по запросу сотрудника, и этот запрос получает уникальный ссылочный идентификатор для дальнейшего использования другими службами. Сотрудник может использовать этот идентификатор, чтобы проверить состояние своего запроса позже. Когда начинается процесс обработки, он вызывает указанную службу для получения идентификатора, который отправляется в качестве ответа на запрос. После этого шага процесс ожидает утверждения заявления (или отказа в утверждении) от менеджера. Время ожидания решения настраиваемо. Менеджер использует другое приложение, в котором он должен ввести идентификатор запроса об отпуске и свое решение. Это действие приводит к вызову того же экземпляра уже запущенного процесса в качестве сигнала, разрешающего продолжение работы. Когда этот сигнал получен процессом, тот продолжает работу и отправляет заявление об отпуске другой службе — части приложения отдела кадров, где ожидаются только утвержденные заявления. Этот шаг завершает процесс, и сотрудник получает долгожданный отдых. Если же менеджер не утверждает и не отклоняет запрос в течение одной недели, процесс посылает письмо с напоминанием и снова ждет утверждения или отклонения заявления от менеджера. Это может повторяться три раза. Если менеджер не утверждает и не отклоняет заявление в течение трех недель, процесс завершается.

Настройка решения

Данное решение требует создания множества проектов в одном решении. Начнем с создания решения `TheHolidayRequestProcessSolution`.

В это решение требуется добавить восемь проектов.

- `HolidayRequestDataContracts`. Библиотека классов, содержащая контракты данных WCF, необходимые для этого решения.
- `HolidayRequestProcess`. Библиотека рабочего процесса, которая содержит определение процесса запроса об отпуске, объявленное как XAML. Этот шаблон можно выбрать в разделе рабочего процесса установленных шаблонов.
- `EmployeeHolidayRequestApplication`. Приложение Windows Forms. Используется сотрудником для подачи заявления об отпуске и запуска процесса его обработки.
- `CalculateReferenceIDService`. Приложение службы WCF. Эта веб-служба выдает уникальный ссылочный идентификатор. Соответствующий шаблон можно найти в веб-разделе установленных шаблонов.
- `ManagersHolidayRequestApprovalApplication`. Приложение Windows Forms, которое позволяет менеджеру принять решение об утверждении или отклонении заявления с соответствующим идентификатором.
- `HolidayRequestProcessHost`. Консольное приложение для размещения процесса, который предоставляет весь процесс обработки заявления в виде службы.
- `HolidayRequestActivityLibrary`. Еще одна библиотека рабочего процесса, которая содержит ссылки на другие службы и позволяет рабочему процессу вызывать методы этих служб. Эти действия используются в библиотеке `HolidayRequestProcess`.
- `ReceiveApprovedHolidayRequestsService`. Консольное приложение для размещения службы, получающей запрос об утверждении. В данном случае она имитирует реальную веб-службу отдела кадров.

Полное решение должно иметь вид, представленный на рис. 13.2.

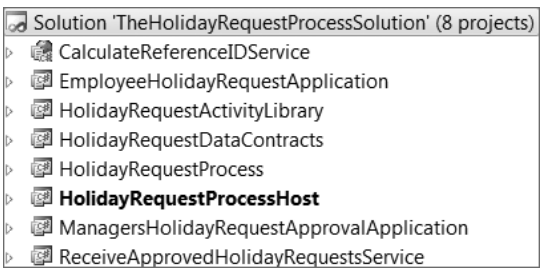


Рис. 13.2. `TheHolidayRequestProcessSolution`

Создание контрактов данных

Каждое использующее WCF решение начинается с определения контрактов данных. Это основа для определения других частей решения. Ясно, что нам требуется стабильное множество контрактов данных перед тем, как продолжить разработку. Мы создадим файл класса в проекте `HolidayRequestDataContracts`, который содержит все необходимые контракты.

В данном случае имеется четыре типа контрактов данных.

- Входной контракт для запроса отпуска, содержащий `EmployeeID`, начальную и конечную даты предполагаемого отпуска.

- Контракт ответа на этот запрос, содержащий уникальный ссылочный идентификатор, вычисляемый внешней службой.
- Контракт для утверждения или отклонения заявления об отпуске менеджером, который содержит идентификатор служащего данного менеджера, уникальный ссылочный идентификатор запроса об отпуске и флаг, указывающий, был ли запрос утвержден или отклонен.
- Контракт ответа на этот запрос об утверждении. В данном случае снова содержит начальную и конечную даты предполагаемого отпуска.

Сначала надо добавить в проект `HolidayRequestDataContracts` ссылку на библиотеку `System.Runtime.Serialization`, чтобы воспользоваться атрибутами для объявления классов в качестве контрактов данных WCF.

Добавьте файл класса в проект, назовите его `HolidayRequestDataContract.cs` и внесите в него следующий код.


```
[DataContract]
public class HolidayRequestDataContract_Input
{
    [DataMember]
    public int EmployeeID { get; set; }
    [DataMember]
    public DateTime HolidayStartDate { get; set; }
    [DataMember]
    public DateTime HolidayEndDate { get; set; }
}

[DataContract]
public class HolidayRequestDataContract_Output
{
    [DataMember]
    public int ReferenceID { get; set; }
}

[DataContract]
public class HolidayApprovalDataContract_Input
{
    [DataMember]
    public int ManagerID { get; set; }
    [DataMember]
    public int ReferenceID { get; set; }
    [DataMember]
    public ApprovedOrDeniedEnum ApprovedOrDenied { get; set; }
}

[DataContract]
public enum ApprovedOrDeniedEnum
{
    [EnumMember]
    Approved,
    [EnumMember]
    Denied
}
```

```
[DataContract]
public class HolidayApprovalDataContract_Output
{
    [DataMember]
    public int EmployeeID { get; set; }
    [DataMember]
    public DateTime HolidayStartDate { get; set; }
    [DataMember]
    public DateTime HolidayEndDate { get; set; }
}
```

Файл CreatingtheBusinessProcess.zip

Создание CalculateReferenceIDService

Эта внешняя служба вычисляет ссылочный идентификатор для запроса об отпуске. Visual Studio добавляет службу `Service1.svc` с файлом кода `Service1.svc.cs` и интерфейсом `IService1.cs`. Можно удалить эти три файла, выполнив следующие действия.

- Добавить новую службу WCF `CalculateReferenceIDService.svc` (рис. 13.3).
- В файле `ICalculateReferenceIDService.cs` удалить операцию `DoWork`, сгенерированную шаблоном вместе с атрибутом `OperationContract`.
- Заменить ее сигнатурой для метода `GetNewReferenceID`.

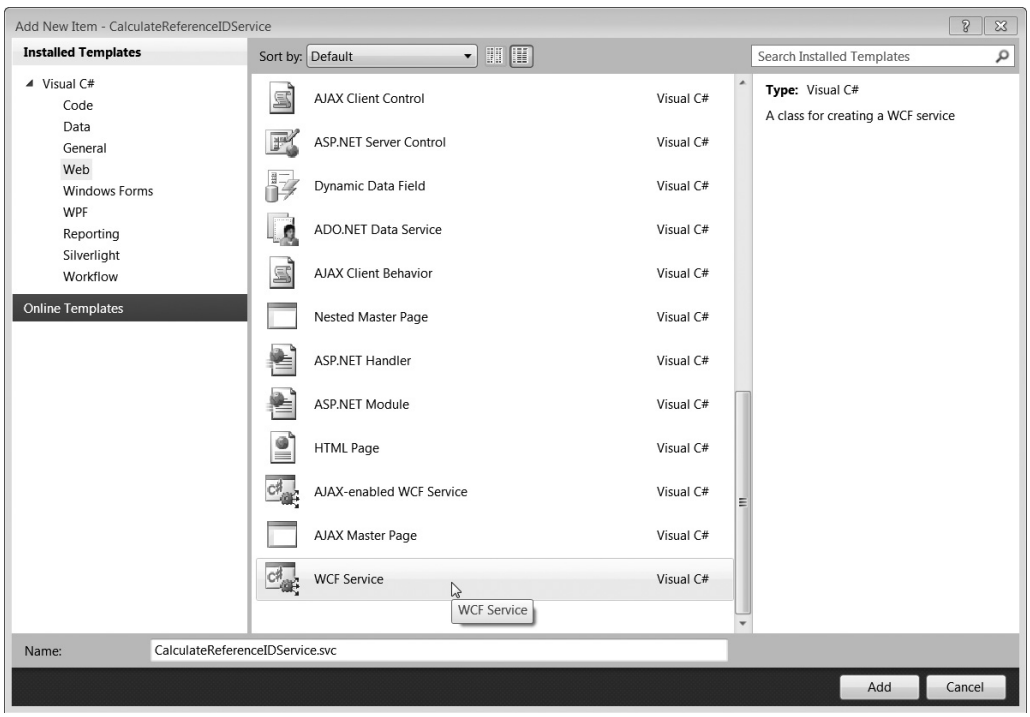


Рис. 13.3. Создание `CalculateReferenceIDService`

```
[ServiceContract]
public interface ICalculateReferenceIDService
{
    [OperationContract]
    int GetNewReferenceID();
}
```

В файле `CalculateReferenceIDService.svc.cs` также можно удалить метод `DoWork` и заменить его реализацией вашего метода `GetNewReferenceID`.

Для нашей реализации, в которой нас интересуют только концептуально связанные с WCF вещи, мы не будем добавлять сложную логику вычисления идентификатора. Чтобы процесс заработал, достаточно вернуть некое целое число, что мы и сделаем. Для проверки работоспособности этого должно хватить. Позже вам придется самостоятельно добавить логику вычисления уникального идентификатора, например, используя для этой цели базу данных.

```
public int GetNewReferenceID()
{
    return 42;
}
```

Сконфигурируем свойства проекта таким образом, чтобы служба работала с конкретным портом, а не с автоматически назначаемым. Это позволит процессу всегда находить службу на одном и том же порту.

Откройте свойства проекта `CalculateReferenceIDService`, перейдите на вкладку Web, выберите Specific port и введите номер порта **`9876`** (рис. 13.4). Можно проверить, корректно ли служба обработала метаданные. В окне Solution Explorer щелкните правой кнопкой мыши на файле `CalculateReferenceIDService.svc` и выберите команду меню View in Browser.

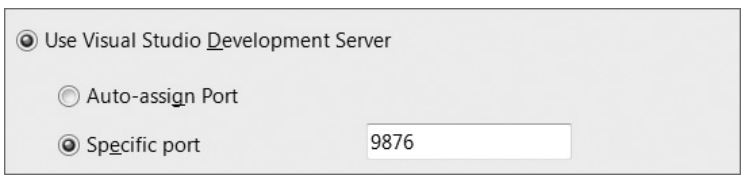


Рис. 13.4. Ввод номера порта

При этом в браузере будет открыт файл `CalculateReferenceIDService.svc`. Щелкните на ссылке `http://localhost:9876/CalculateReferenceIDService.svc?wsdl`. Теперь вы должны увидеть WSDL для данной службы. Если браузер выводит только содержимое каталога, щелкните на ссылке `CalculateReferenceIDService.svc`.

Создание `ReceiveApprovedHolidayRequestService`

Приложение `ReceiveApprovedHolidayRequestService` создается как консольное, а не как веб-приложение. Это делается для удобства тестирования, поскольку в таком приложении проще показать результаты работы тестеру. В данном консольном приложении совместно представлены код для открытия хоста, объявления интерфейса

службы и его реализации, что делает его "самообслуживающим" хостом. Причина этого в том, чтобы можно было видеть на консоли результаты вызова операций службы. Что весьма удобно в дидактических целях, но промышленный вариант может использовать службу IIS с теми же контрактом и конфигурацией.

Добавьте ссылки на библиотеки `System.ServiceModel` и `System.Runtime.Serialization`.

Добавьте файл класса `ReceiveApprovedHolidayRequestService.cs` и внесите в него следующий код.


```
[ServiceContract]
public interface IReceiveApprovedHolidayRequestService
{
    [OperationContract]
    void ReceiveApprovedHolidayRequest(
            ApprovedHolidayData approvedHolidayData);
}

public class ReceiveApprovedHolidayRequestService :
IReceiveApprovedHolidayRequestService
{
    public void ReceiveApprovedHolidayRequest
    (ApprovedHolidayData approvedHolidayData)
    {
        Console.WriteLine("Got Approved Holiday Request");
        Console.WriteLine(
        string.Format(" by employee {0}, approved by {1},"
                      " from {2} to {3}",
        approvedHolidayData.EmployeeID.ToString(),
        approvedHolidayData.ApprovedByManagerID.ToString(),
        approvedHolidayData.HolidayStartDate.ToShortTimeString(),
        approvedHolidayData.HolidayEndDate.ToShortTimeString()));
    }
}

[DataContract]
public class ApprovedHolidayData
{
    [DataMember]
    public int EmployeeID { get; set; }
    [DataMember]
    public int ApprovedByManagerID { get; set; }
    [DataMember]
    public DateTime HolidayStartDate { get; set; }
    [DataMember]
    public DateTime HolidayEndDate { get; set; }
}
```

Файл CreatingtheBusinessProcess.zip

Теперь файл содержит контракт данных, контракт службы и реализацию операции.

Добавьте в проект файл конфигурации приложения, `app.config`, который должен содержать следующее.


```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="ExposeMetadata">
        <serviceMetadata
          httpGetEnabled="true"
          httpGetUrl="http://localhost:9875/
↳           ReceiveApprovedHolidayRequestsService/MEX"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service
        name="ReceiveApprovedHolidayRequestsService.
↳             ReceiveApprovedHolidayRequestService"
        behaviorConfiguration="ExposeMetadata">
      <endpoint
          address="http://localhost:9875/
↳            ReceiveApprovedHolidayRequestsService"
          binding="basicHttpBinding"
          bindingConfiguration=""
          contract="ReceiveApprovedHolidayRequestsService.
↳            IReceiveApprovedHolidayRequestService" />
    </service>
  </services>
</system.serviceModel>
```

Файл CreatingtheBusinessProcess.zip

Для размещения службы добавьте в основной метод файла `program.cs` следующий код.

```
try
{
    Console.WriteLine(
        "HOST : ReceiveApprovedHolidayRequestsService");
    ServiceHost serviceHost;
    serviceHost =
        new ServiceHost(typeof(
            ReceiveApprovedHolidayRequestService));
    serviceHost.Open();
    Console.WriteLine("started");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
Console.ReadKey();
```

Теперь можно протестировать это консольное приложение, чтобы убедиться в корректной обработке данных WSDL. Сначала следует собрать приложение и запустить его с правами администратора вне Visual Studio. Откройте каталог, в котором находится выполнимый файл, щелкните на нем правой кнопкой мыши и выберите пункт меню Run As Administrator.

Откройте браузер и введите в нем URL, указанный в элементе `serviceMetadata` элемента `behavior` конфигурационного файла, а именно `http://localhost:9875/ReceiveApprovedHolidayRequests Service/MEX`.

В браузере вы должны увидеть данные WSDL.

Добавление ссылок на службы в проект HolidayRequestActivityLibrary

Проект `HolidayRequestActivityLibrary` содержит только две ссылки на только что созданные службы, и больше ничего. Поэтому файл `Activity1.xaml` можно смело удалить.

Добавление ссылок на службы приводит к созданию повторно используемых действий по вызову операций этих служб. Эти действия становятся видимы проектировщику рабочего процесса или другим библиотекам, которые обращаются к данному проекту, как, например, проект `HolidayRequestProcess`. Это хороший подход к проектированию, отделяющий действия, представляющие собой оболочки для вызовов служб, от самого рабочего процесса.

В результате это приводит к повторному использованию действий в других бизнес-процессах.

Добавление ссылок на службы генерирует также конфигурацию для конечных точек клиента в файле `app.config`. Здесь эта конфигурация не используется, поскольку проект представляет собой библиотеку. Как вы увидите позже, необходимо скопировать эту конфигурацию в проект, служащий для размещения рабочего процесса.

Добавление CalculateReferenceIDService

Чтобы добавить ссылку на `CalculateReferenceIDService`, щелкните правой кнопкой мыши на References и выберите Add Service Reference (рис. 13.5). Откроется диалоговое окно Add Service Reference.

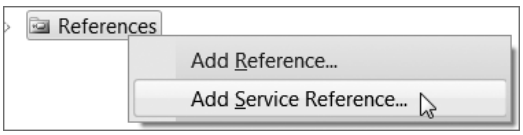


Рис. 13.5. Добавление ссылки на службу

Щелкните на кнопке Discover и выберите Services in Solution (рис. 13.6). В результате будет получен обзор интерфейсов, содержащих операции этой службы.

Вы увидите операцию `GetNewReferenceID` (рис. 13.7). Установите пространство имен `CalcIDService`.

Щелкните на кнопке OK, после чего Visual Studio сгенерирует прокси и добавит в конфигурационный файл конечною точку.

Добавление ReceiveApprovedHolidayRequestsService

Добавьте вторую ссылку на службу к `ReceiveApprovedHolidayRequestsService`. Visual Studio не в состоянии обнаружить решения WSDL, поскольку это служба, размещенная в самом Visual Studio. Следует запустить `ReceiveApprovedHolidayRequests-Service` извне Visual Studio, причем от имени администратора.

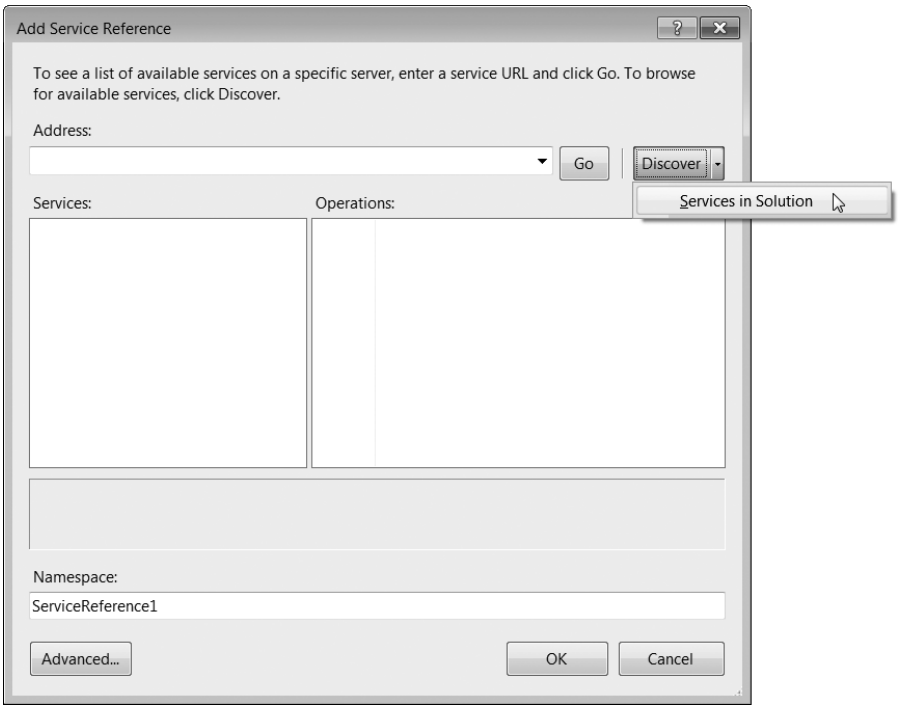


Рис. 13.6. Выбор службы из решения

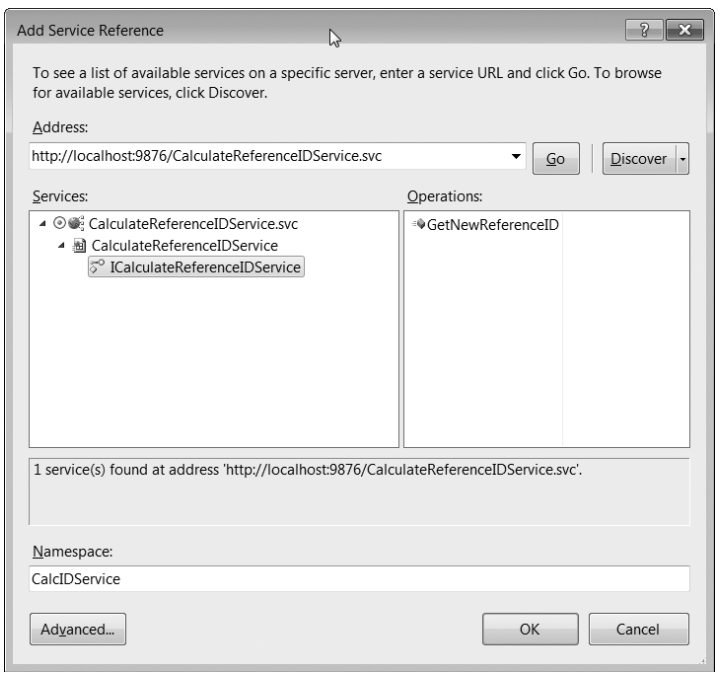


Рис. 13.7. Установка пространства имен `CalcIDService`

Адресом данной службы является

```
http://localhost:9875/ReceiveApprovedHolidayRequestsService/MEX
```

В качестве пространства имен для ссылки на службу используйте `HRService`.

Посмотрим на файл `app.config` проекта `HolidayRequestActivityLibrary`. Он должен содержать две конечные точки.

```
<endpoint
  address=
    "http://localhost:9876/CalculateReferenceIDService.svc"
  binding="basicHttpBinding"
  bindingConfiguration=
    "BasicHttpBinding_ICalculateReferenceIDService"
  contract="ICalculateReferenceIDService"
  name="BasicHttpBinding_ICalculateReferenceIDService" />
<endpoint
  address=
    "http://localhost:9875/ReceiveApprovedHolidayRequestsService"
  binding="basicHttpBinding"
  bindingConfiguration=
    "BasicHttpBinding_IReceiveApprovedHolidayRequestService"
  contract="IReceiveApprovedHolidayRequestService"
  name=
    "BasicHttpBinding_IReceiveApprovedHolidayRequestService" />
```

Теперь можно закрыть консольное приложение с размещенной службой `ReceiveApprovedHolidayRequestsService`.

Разработка `HolidayRequestProcess`

Процесс `HolidayRequestProcess` содержит определение рабочего процесса и ссылки на ряд системных библиотек: `System.Activities`, `System.Runtime.Serialization`, `System.ServiceModel` и `System.ServiceModel.Activities`.

Этому проекту требуются также ссылки на библиотеки `HolidayRequestActivityLibrary` и `HolidayRequestDataContracts`. Первая нужна постольку, поскольку библиотека содержит действия для вызова операций внешних служб (`CalcIDService` и `HRService`), а `HolidayRequestDataContracts` содержит контракты данных, которые определяют входные и выходные параметры для операций служб, которые вы разрабатываете в рабочем процессе.

В службе рабочего процесса создаются две операции, `RequestHoliday` и `ApproveRequest`, что делается путем перетаскивания действий в проектировщик рабочего процесса. Данный подход позволяет определить контракт службы при разработке бизнес-процесса.

Добавление рабочего процесса

Добавьте действие рабочего процесса в проект. Для этого выберите шаблон в части Workflow диалогового окна Add New Item (рис. 13.8). Назовите это действие рабочего процесса `HolidayRequestProcessDefinition.xaml`.

Соберите полное решение, откройте действие и сделайте видимой инструментальную панель. Эта панель теперь содержит два действия в качестве оберток для операций двух служб (рис. 13.9). В настоящий момент поверхность проектировщика пуста. Добавьте действие `ReceiveAndSendReply` из панели в проектировщик.

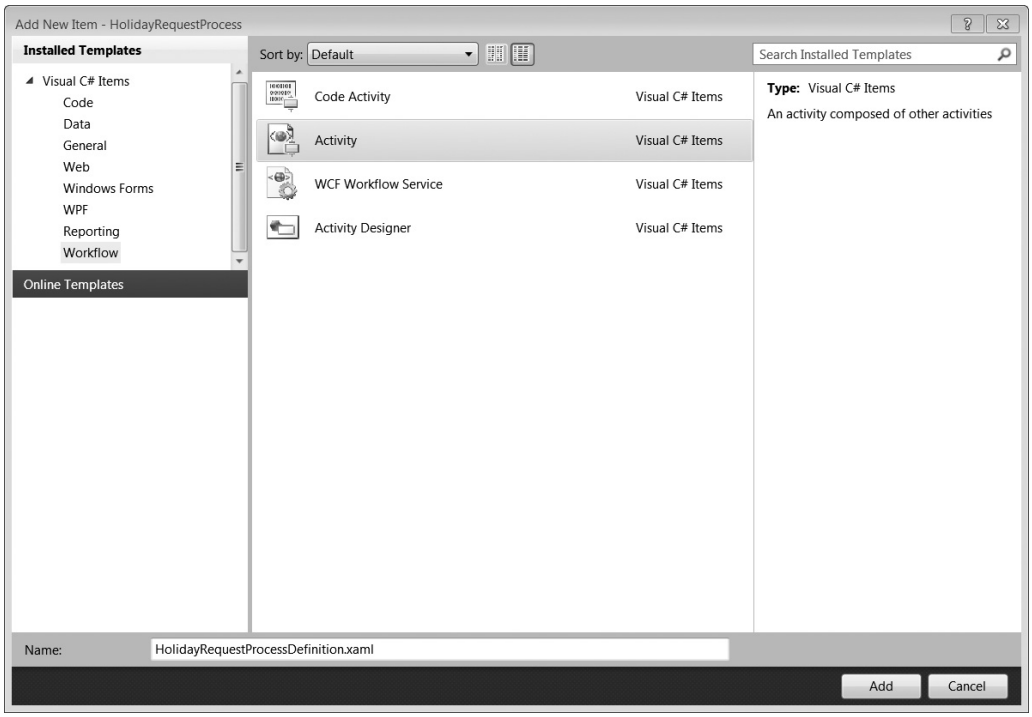


Рис. 13.8. Добавление действия рабочего процесса в проект

Рис. 13.9. Панель с двумя действиями

Это действие теперь может быть найдено в разделе Messaging диалогового окна Toolbox (рис. 13.10). Просто перетащите действие в Drop Activity Here.

В результате получим два отдельных действия (рис. 13.11).

- Первое – действие `Receive`, которое определяет входные параметры операции.
- Второе – действие `SendReply`, которое определяет возвращаемый тип операции.

Создание переменных

Перед детальной настройкой действий для рабочего процесса необходимо создать переменные для хранения входных данных, полученных от действий, а также данных, которые будут отправлены назад в качестве ответа.

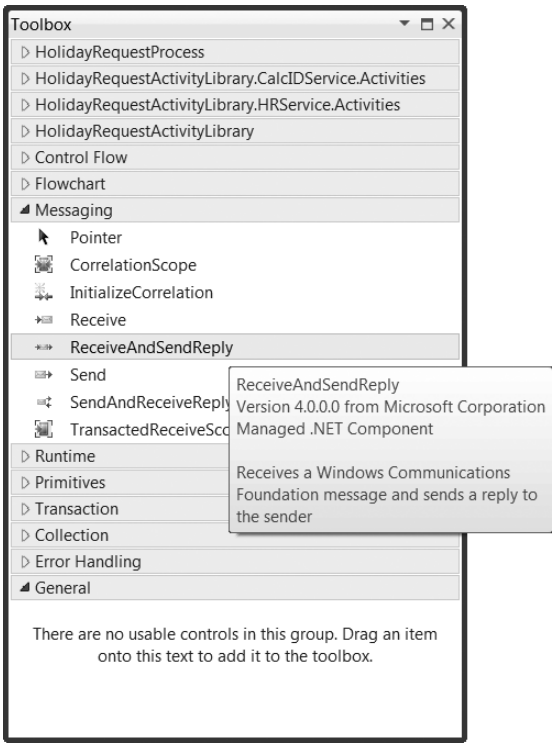


Рис. 13.10. Действие `ReceiveAndSendReply`

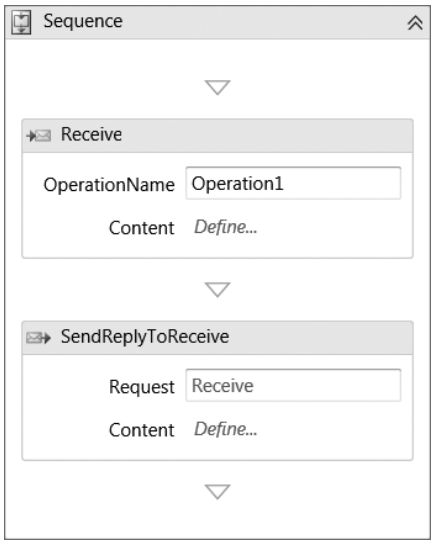


Рис. 13.11. Действия в проектировщике

Необходима также переменная для хранения уникального ссылочного идентификатора, полученного от внешней службы. Нужна и переменная для количества выполненных итераций ожидания тайм-аута.

Добавьте шесть переменных — для их создания щелкните на последовательности действий для установки области видимости, а также на кнопке Variables в нижней левой части проектировщика, как показано на рис. 13.12. Каждая переменная требует указания типа. Эти типы либо определены в библиотеке `HolidayRequestDataContracts`, либо генерируются прокси в библиотеке `HolidayRequestActivityLibrary`, либо просто представляют собой целочисленное значение для ссылочного идентификатора и `retryCounter`.

Рис. 13.12. Вкладка Variables

Тип переменной выбирается в диалоговом окне Browse and Select a .NET Type, которое можно открыть, выбрав пункт Browse for Types в столбце Variable type списка переменных (рис. 13.13).

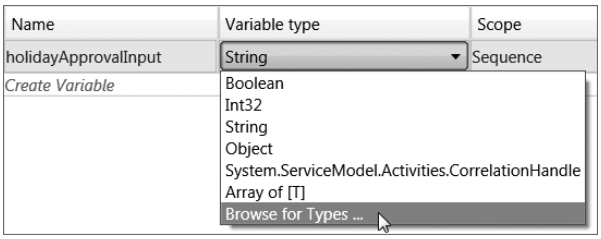


Рис. 13.13. Просмотр типов

Создайте переменные `holidayApprovalInput`, `holidayApprovalOutput`, `holidayRequestInput`, `holidayRequestOutput`, `approvedHolidayInput`, `referenceID` и `retryCounter`. Выберите для каждой переменной в диалоговом окне соответствующий тип (рис. 13.14).

Ниже приведены все переменные и их типы.

- `holidayApprovalInput`: `HolidayApprovalDataContract_Input`
- `holidayApprovalOutput`: `HolidayApprovalDataContract_Output`
- `holidayRequestInput`: `HolidayRequestDataContract_Input`
- `holidayRequestOutput`: `HolidayRequestDataContract_Output`
- `referenceID`: `Int32`
- `approvedHolidayInput`: `ApprovedHolidayData`
- `correlationHandle`: `CorrelationHandle`
- `retryCounter`: `Int32`

Результат должен выглядеть так, как показано на рис. 13.15.

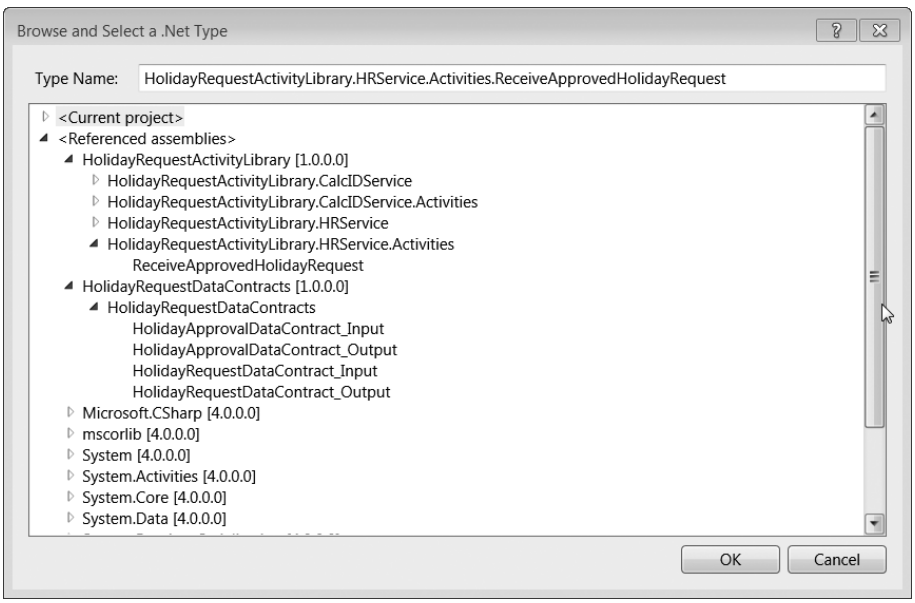


Рис. 13.14. Диалоговое окно Browse and Select a .Net Type

Name	Variable type	Scope
holidayApprovalInput	HolidayApprovalDataContract_Input	Sequence
holidayApprovalOutput	HolidayApprovalDataContract_Output	Sequence
holidayRequestInput	HolidayRequestDataContract_Input	Sequence
holidayRequestOutput	HolidayRequestDataContract_Output	Sequence
referenceID	Int32	Sequence
approvedHolidayInput	ApprovedHolidayData	Sequence
correlationHandle	CorrelationHandle	Sequence
retryCounter	Int32	Sequence

Рис. 13.15. Все переменные

Настройка действия `Receive`

Измените значение Operation Name на `RequestHoliday` (рис. 13.16). Для определения параметров этой операции щелкните на поле Define.

В открывшемся диалоговом окне Content Definition установите переключатель Parameters и щелкните на ссылке Add New Parameter. Назовите параметр — `inputparam_holidayRequest`. Для определения его типа выберите в выпадающем списке Browse for Types. Интересующий нас тип — `HolidayRequestDataContract_Input` в пространстве имен `HolidayRequestDataContracts`.

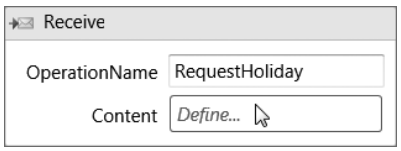


Рис. 13.16. Действие `Receive`

Вернитесь к диалоговому окну Content Definition и введите имя переменной для входных данных — `holidayRequestInput` (рис. 13.17).

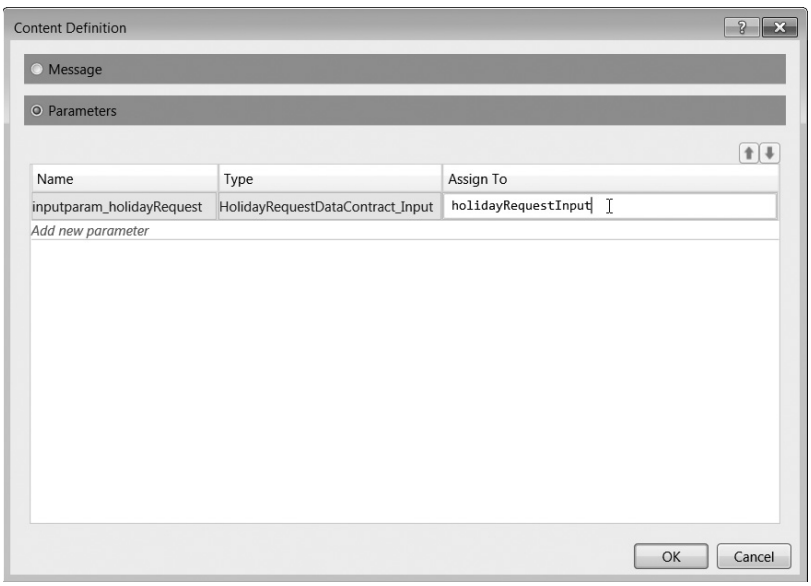


Рис. 13.17. Диалоговое окно Content Definition

Обратите внимание на то, что это поле ввода поддерживает возможность IntelliSense и позволяет быстро найти имя переменной.

Закройте диалоговое окно Content Definition.

Вернитесь к проектировщику, откройте окно свойств действия `Receive` и установите флаг свойства `CanCreateInstance`. Этим вы указываете, что вызов данной операции создает новый экземпляр рабочего процесса (рис. 13.18).

Настройка действия `Send`

Перед тем как настраивать действие `Send`, надо добавить переменную связи. Эта переменная используется в конфигурации действия `SendReply` и позже в действии `Receive` для операции `ApproveRequest`. Вновь активизируйте в проектировщике последовательность для установки области видимости и откройте список переменных.

Добавьте переменную `correlationHandle` типа `CorrelationHandle` (рис. 13.19) и щелкните на ссылке `Define` в поле ввода Content действия `Send`.

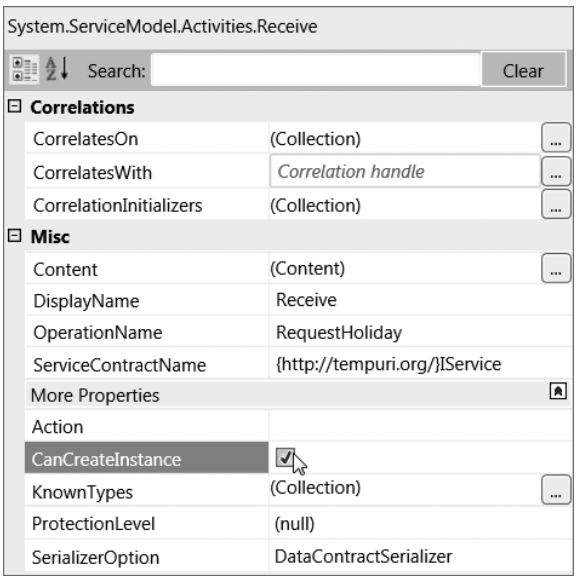


Рис. 13.18. Установите флаг `CanCreateInstance`

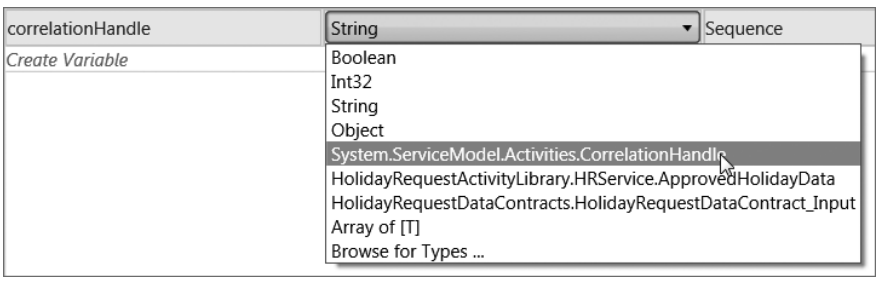


Рис. 13.19. Выбор `correlationHandle` в качестве типа

В открывшемся окне Content Definition выберите параметры, добавьте новый параметр `outputparam_holidayRequest` и установите его тип — `HolidayRequest-DataContract_Output`, — используя Browse for Types, что открывает диалоговое окно Select a .NET Type. Установите значение переменной `holidayRequest-Output`. В проектировщике откройте окно свойств действия `Send` и откройте окно Add Correlation Initializers щелкнув на кнопке с многоточием рядом со свойством `CorrelationInitializers` (рис. 13.20).

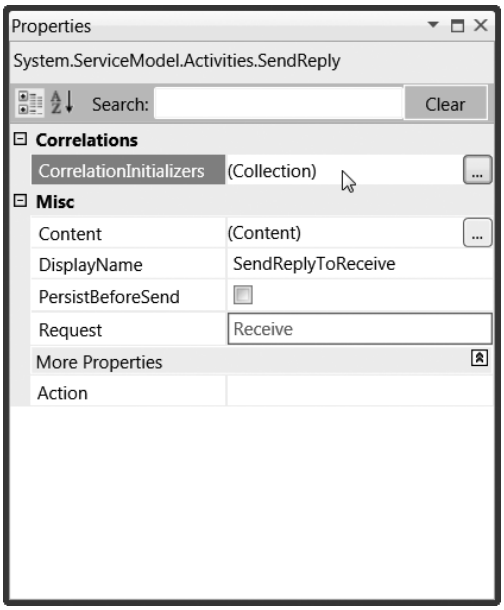


Рис. 13.20. Свойства `SendReply`

В этом диалоговом окне щелкните на ссылке Add Initializer и введите `correlationHandler`. Это — имя переменной, создаваемой для связи действия `Send` с действием `Receive` операции `ApproveRequest`. Теперь перейдите к раскрывающемуся списку XPATH Queries и выберите в нем `ReferenceID` из раздела `outputparam_holidayRequest` (рис. 13.21).

Между действиями `Receive` и `Send` необходимо вызвать `CalcIDService`, чтобы получить ссылочный идентификатор, и присвоить его значение значению, возвращаемому операцией.

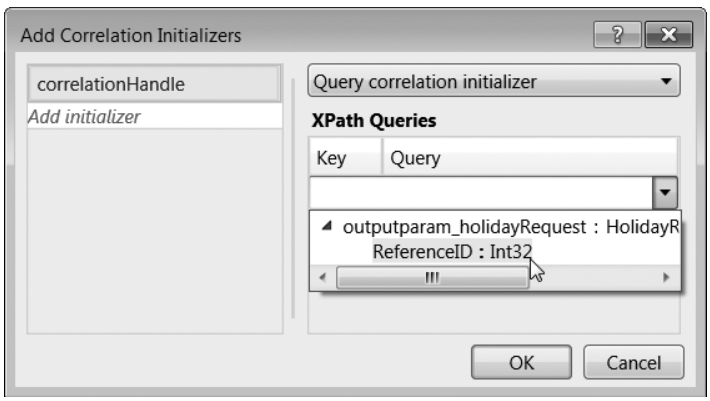


Рис. 13.21. Инициализатор связи

Перетащите действие `GetNewReferenceID` из панели инструментов и отпустите между действиями `Receive` и `Send`.

Откройте окно свойств этого действия и щелкните на кнопке с многоточием напротив свойства `GetNewReferenceIDResult`. Это приведет к открытию Expression Editor. В этом редакторе введите `referenceID`. Это имя переменной, которая хранит `referenceID` (рис. 13.22). Теперь надо присвоить полученное значение `referenceID` выходному параметру действия `Send`. Для этого надо перетащить два действия `Assign`

между действиями GetNewReferenceID и SendActivity. Действие Assign можно найти в инструментальной панели в части Primitives. Первое действие Assign инстанцирует переменную holidayRequestOutput, а второе присваивает значение referenceID выходному значению действия GetNewReferenceID.

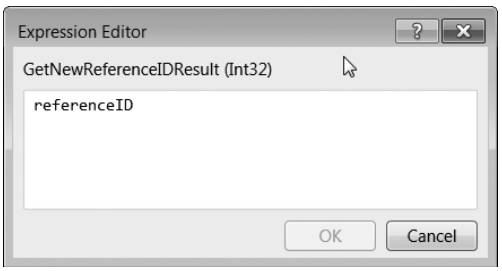


Рис. 13.22. Редактор выражения

Откройте окно Properties первого действия Assign и настройте его свойства (рис. 13.23).

- To: holidayRequestOutput
- Value: New HolidayRequestDataContract_Output()

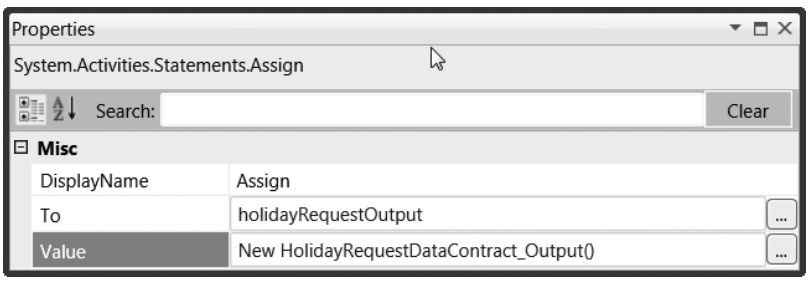


Рис. 13.23. Свойства действия Assign

Откройте окно Properties второго действия Assign и настройте его свойства (рис. 13.24).

- To: holidayRequestOutput.ReferenceID
- Value: referenceID

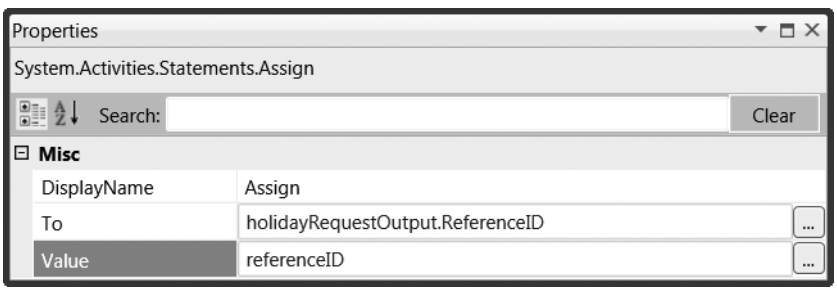


Рис. 13.24. Свойства действия Assign

Добавьте действие `WriteLine` после действия `Send`, чтобы увидеть, когда вызывается операция `RequestHoliday` и чему равно получаемое от службы значение `referenceID`. Для этого перетащите действие `WriteLine` из части Primitives инструментальной панели (рис. 13.25).

Рис. 13.25. Действие `WriteLine` в панели инструментов

Действие `WriteLine` в данном случае предназначено для отладочных целей. Оно выводит строку данных в консольном приложении, которое служит для размещения этой службы рабочего процесса. В окончательной версии программы это действие можно удалить.

Откройте окно Properties этого действия и настройте свойство Text так, чтобы оно содержало отформатированную строку, выводящую полученное значение `referenceID`. Его можно настроить в Expression Editor, щелкнув для вызова последнего на соответствующей кнопке с многоточием. Добавьте в редактор код, показанный на рис. 13.26.

```
String.Format("Received Request {0}", referenceID)
```

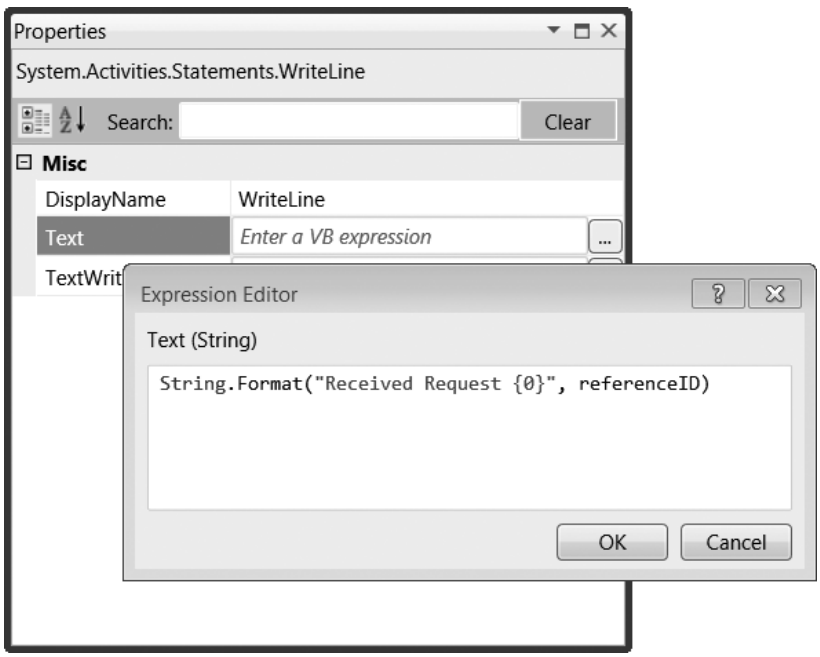


Рис. 13.26. Редактор выражения

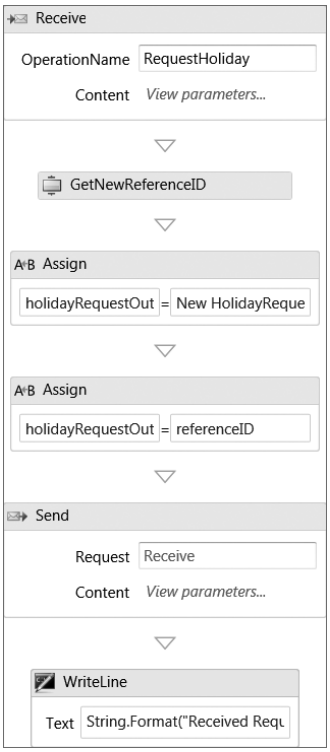


Рис. 13.27. Завершенная последовательность

Теперь последовательность должна выглядеть так, как показано на рис. 13.27.

Следующий шаг процесса — добавление действия `while`. Его можно найти в части Control Flow панели инструментов.

Действие `while` позволяет определить и выполнять некоторые шаги, пока истинно указанное условие. В данном случае это условие основано на переменной `retryCounter`. Шаги повторяются, пока значение `retryCounter` меньше трех. Счетчик `retryCounter` увеличивается действиями, входящими в действие `while`.

Откройте окно Properties действия `while` и установите значение свойства Condition как логическое, представляющее собой выражение `retryCounter<3` (рис. 13.28).

Следующий шаг состоит в добавлении действия `Pick` в теле действия `while`. Действие `Pick` позволяет процессу ожидать триггер перед тем, как продолжить выполнение. В данном случае процесс должен ждать либо истечения тайм-аута, указанного действием `Delay`, либо получения утверждения или отказа от менеджера.

Действие `Pick` может быть найдено в разделе Control Flow панели инструментов.

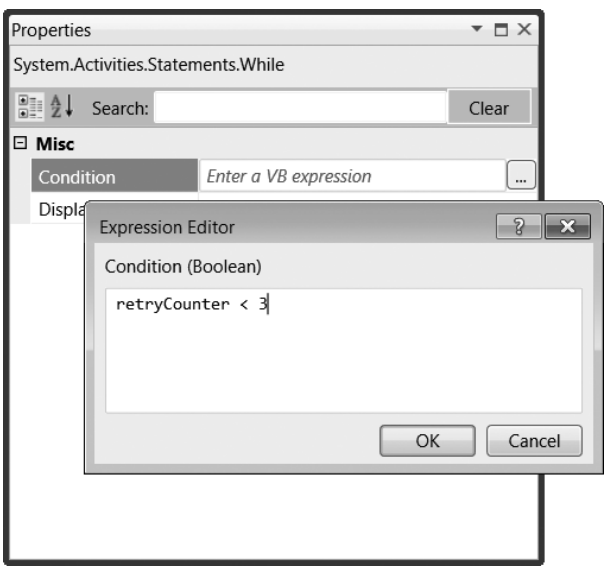


Рис. 13.28. Условие действия `while`

Перетащите действие `Delay` в область Trigger в левой части окна действия `Pick` (рис. 13.29).

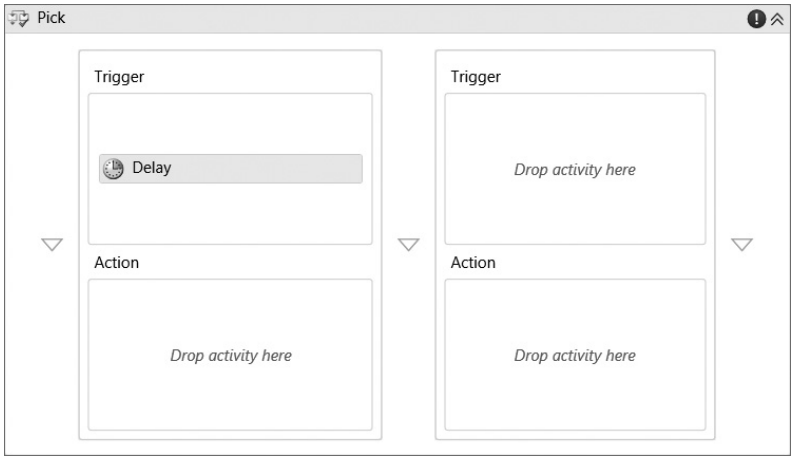


Рис. 13.29. Действие `Pick` в проектировщике

Откройте окно Properties действия `Delay` и настройте свойство длительности щелчком на кнопке с многоточием. Продолжительность определяется как выражение VB.NET. Для теста используем продолжительность 30 секунд, вводя `System.TimeSpan.FromSeconds(30)` (рис. 13.30).

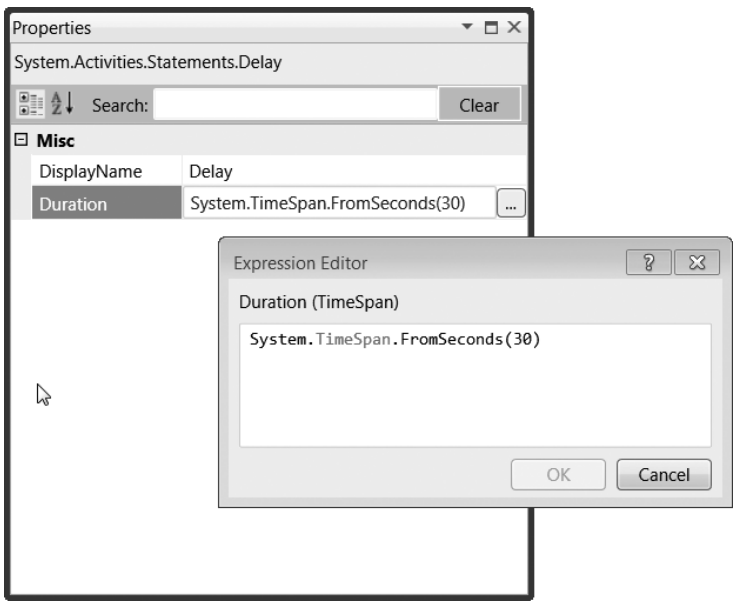


Рис. 13.30. Определение времени задержки

В рабочем варианте следует установить бо́льшую продолжительность, выражаемую в неделях.

В области действия в правой части окна действия `Pick`, ниже задержки, добавьте действие `Sequence`. Это действие состоит из двух. Первое — действие, `WriteLine`, выводит на консоль сообщение об истечении срока задержки. Это временное действие, предназначенное только для того, чтобы проверить, как работает задержка. Второе действие, `Assign`, увеличивает переменную `retryCounter`.

Добавьте действие `WriteLine` и укажите его текст о задержке. Затем добавьте действие `Assign` и установите его свойство `To` равным `retryCounter`, а свойство `Value` равным `retryCounter+1` (рис. 13.31).

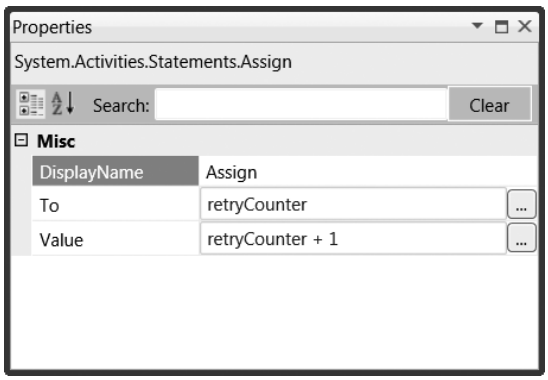


Рис. 13.31. Свойства действия `Assign`

Настройка действия `ReceiveAndSendReply`

В область Trigger последовательности в правой части окна действия `Pick` перетащите другое действие `ReceiveAndSendReply` (рис. 13.32). Это создаст вторую операцию службы рабочего процесса для получения утверждения или отказа запроса на отпуск. Данная операция вызывается приложением `ManagersHolidayRequestApproval`.

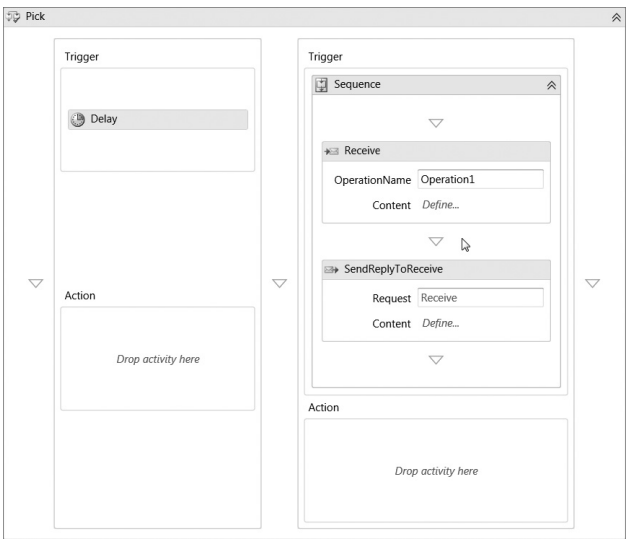


Рис. 13.32. Вид действия `Pick`

Установите значение поля OperationName равным `ApproveRequest`. Щелкните на ссылке Define для настройки содержимого действия `Receive`. В окне Content Definition выберите переключатель Parameters. Добавьте параметр `inputparam_holidayApproval`, установите его тип `HolidayRequestDataContract_Input` и присвойте этот параметр переменной `holidayApprovalInput` (рис. 13.33).

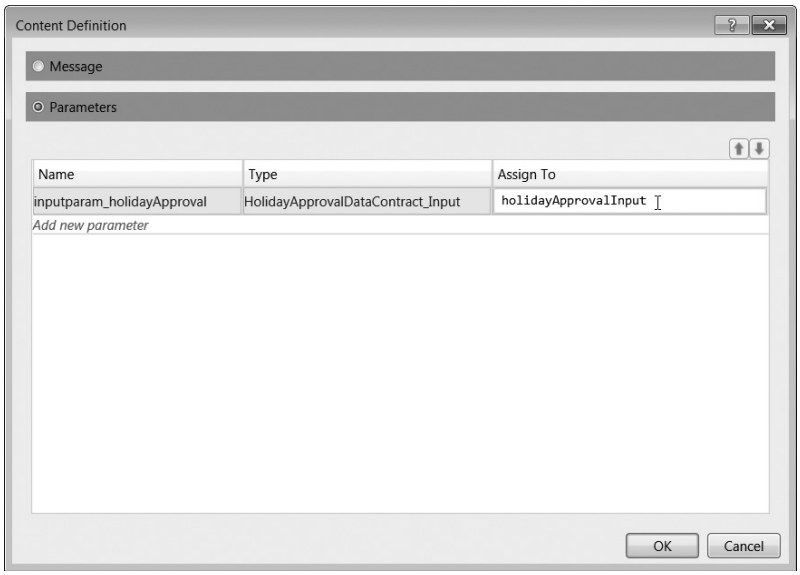


Рис. 13.33. Присваивание параметра

Настройте связь, устанавливая значение `CorrelatesWith` равным `correlationHandle`.

Откройте окно CorrelatesOn Definition, щелкнув на кнопке с многоточием напротив свойства `CorrelatesOn`. Выберите в раскрывающемся списке XPath Queries в качестве параметра для связи `ReferenceID` (рис. 13.34).

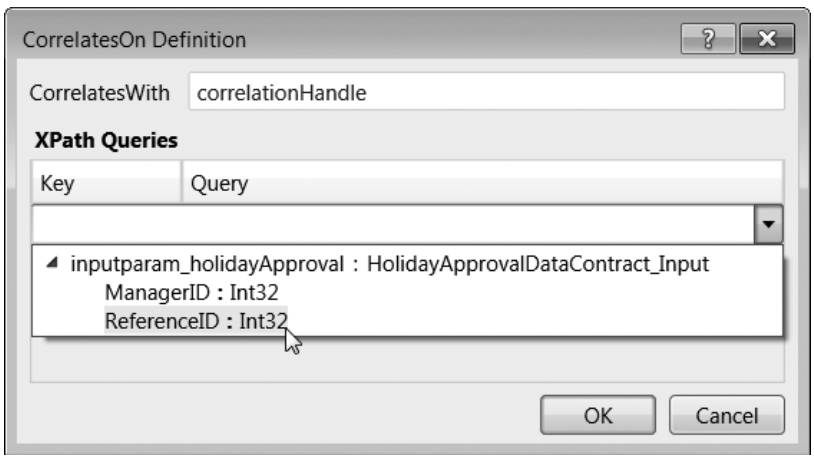


Рис. 13.34. Настройка связи

Настройте действие SendReply. Откройте окно Content Definition, выберите Parameters и добавьте параметр outputparam_holidayApproval. Установите его тип HolidayApprovalDataContract_Output, найденный в HolidayRequestDataContracts, и присвойте его переменной holidayApprovalOutput (рис. 13.35).

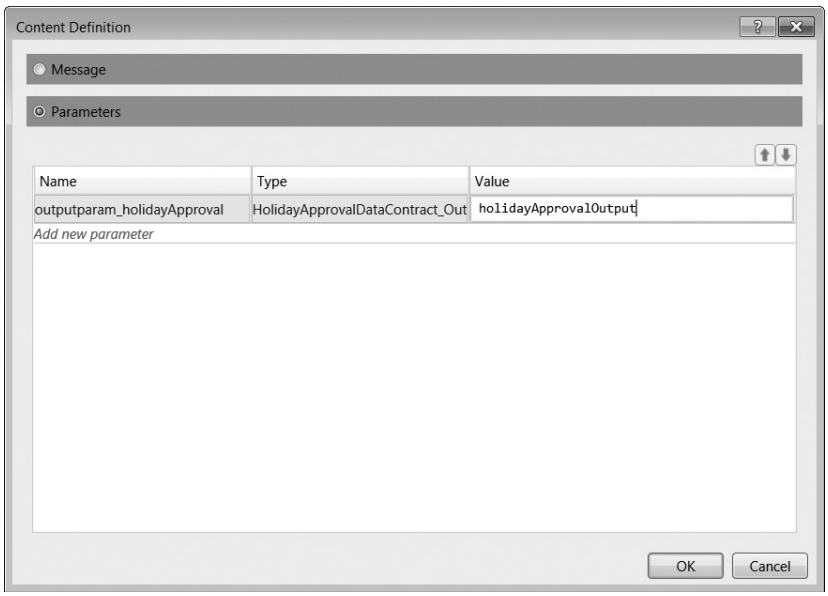


Рис. 13.35. Окно Content Definition

Между действиями Receive и SendReply перетащите еще два действия Assign. Первое действие Assign инстанцирует выходную переменную, а второе присваивает значение EmployeeID, полученное как входное для запроса об утверждении отпуска, в качестве свойства этой переменной.

```
holidayApprovalOutput                New HolidayApprovalDataContract_Output()
holidayApprovalOutput.EmployeeID     holidayRequestInput.EmployeeID
```

После действия SendReply добавьте действие Assign, которое устанавливает значение переменной retryCounter равным 99. Результат должен выглядеть так, как показано на рис. 13.36.

Это значение больше, чем использованное в условии действия while, и приведет к остановке этого действия (рис. 13.37). В части Action окна действия Pick добавьте действие If. Его можно найти в разделе Control Flow панели инструментов. Это действие будет выполнять проверку значения переменной holidayApprovalInput для выяснения, утверждено или отклонено заявление об отпуске.

Измените значение поля DisplayName на ApprovedOrDenied. Отредактируйте условие, введя следующее выражение.

```
holidayApprovalInput.ApprovedOrDenied = ApprovedOrDeniedEnum.Approved
```

Результат показан на рис. 13.38.

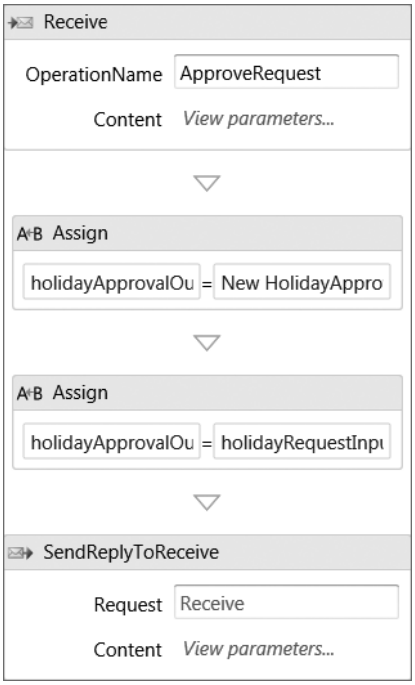


Рис. 13.36. Инструкции приложения

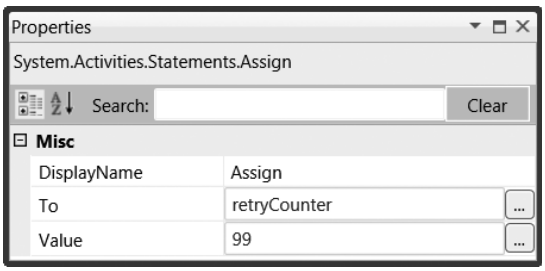


Рис. 13.37. Установка значения `retryCounter`

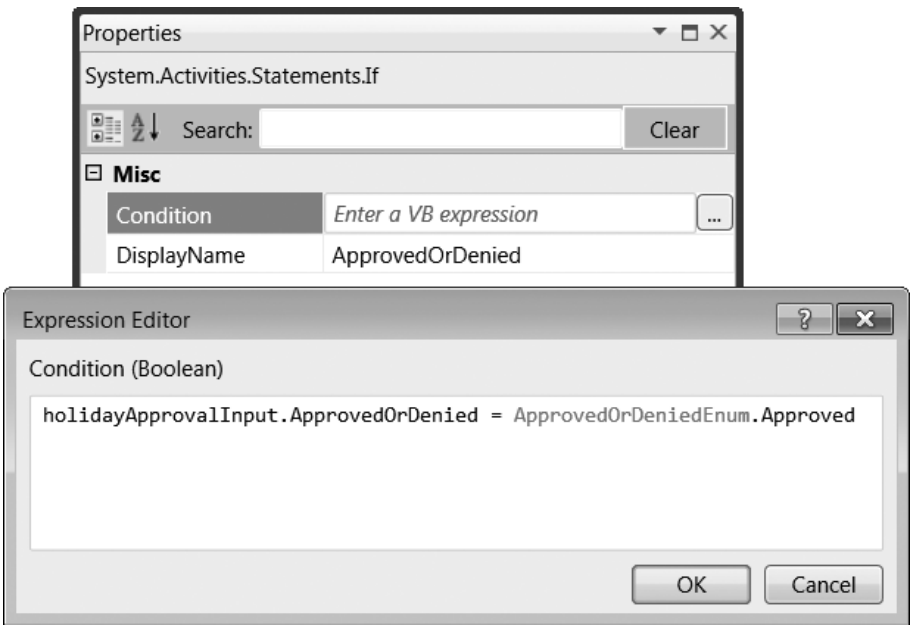


Рис. 13.38. Условие в редакторе выражений

В ветвь Then действия If перетащите действие Sequence для группировки всех действий, необходимых при утверждении заявления об отпуске. Перетащите действие Sequence в ветвь Else.

В левое действие Sequence добавьте из панели инструментов действие ReceiveApprovedHolidayRequest. Вызов HRService будет выполняться только в том случае, когда заявление об отпуске будет утверждено.

Установите свойство approvedHolidayData этого действия равным переменной approvedHolidayInput (рис. 13.39).

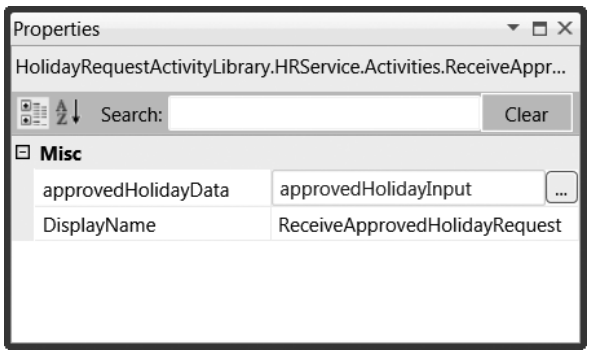


Рис. 13.39. Установка свойства approvedHolidayData

Добавьте следующие пять действий в часть перед действием ReceiveApprovedHolidayRequest. Эти действия создают переменную approvedHolidayInput. Настройте свойства To и Value действий следующим образом (рис. 13.40).

```
To    : approvedHolidayInput
Value: New HolidayRequestActivityLibrary.HRService.ApprovedHolidayData

To    : approvedHolidayInput.ApprovedByManagerID
Value: holidayApprovalInput.ManagerID

To    : approvedHolidayInput.EmployeeID
Value: holidayRequestInput.EmployeeID

To    : approvedHolidayInput.HolidayStartDate
Value: holidayRequestInput.HolidayStartDate

To    : approvedHolidayInput.HolidayEndDate
Value: holidayRequestInput.HolidayEndDate
```

Добавьте действие WriteLine к каждой ветви для вывода на консоль информации о том, какое решение принято менеджером.

Разработка HolidayRequestHost

В проект HolidayRequestHost следует добавить ссылки на HolidayRequestActivityLibrary, HolidayDataContracts и HolidayRequestProcess.

Консольному приложению, кроме того, требуются ссылки на System.Activities, System.ServiceModel, System.Runtime.Serialization, System.ServiceModel.Activities и System.ServiceProcess (рис. 13.41).

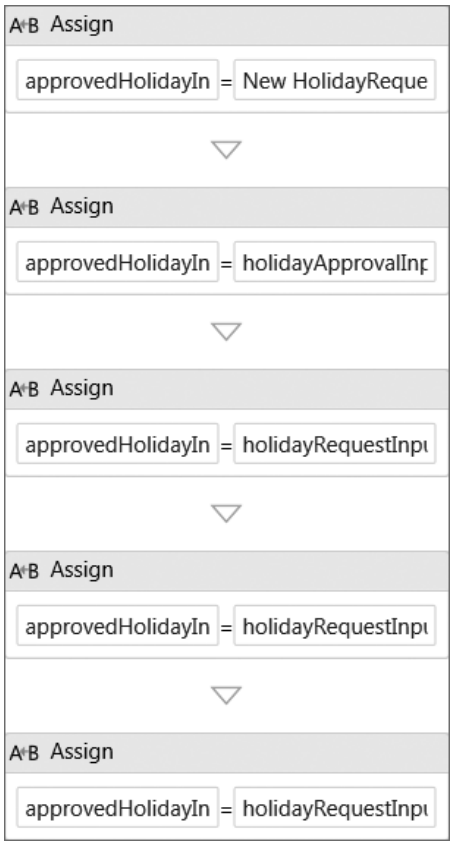


Рис. 13.40. Инструкции присвоения

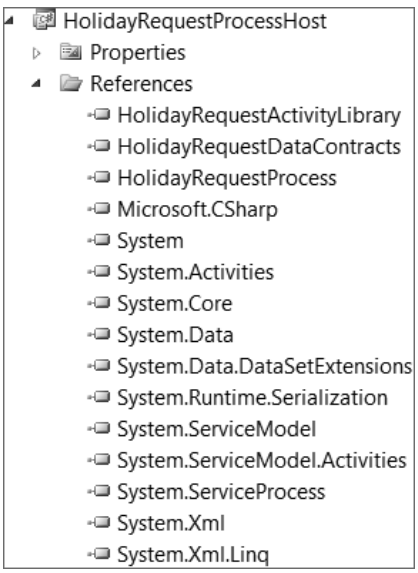


Рис. 13.41. Ссылки, необходимые хосту

Добавьте в проект конфигурационный файл приложения. Скопируйте и вставьте в этот конфигурационный файл полное содержимое файла `app.config` из проекта `HolidayRequestActivityLibrary`.

В файл `program.cs` добавьте в основной метод код для инстанцирования `WorkflowServiceHost`; кроме того, добавьте `SqlWorkflowInstanceStoreBehavior` в `WorkflowServiceHost`, `ServiceMetadataBehavior`, а затем установите интервал `timeToUnload` равным `0` секунд.


```
try
{
  WorkflowServiceHost workflowServiceHost;
  workflowServiceHost = new WorkflowServiceHost(
    new HolidayRequestProcess.
        HolidayRequestProcessDefinition(),
    new Uri(@"http://localhost:9874/HolidayRequestProcess"));

  workflowServiceHost.Description.Behaviors.Add(
    new SqlWorkflowInstanceStoreBehavior(
      "Data Source=.;Initial Catalog=
↳     SqlWorkflowInstanceStore;Integrated Security=True"));
```

```
  ServiceMetadataBehavior serviceMetadataBehavior;
  serviceMetadataBehavior = new ServiceMetadataBehavior();
  serviceMetadataBehavior.HttpGetEnabled = true;
  workflowServiceHost.Description.Behaviors.Add(
        serviceMetadataBehavior);
  WorkflowIdleBehavior workflowIdleBehavior =
        new WorkflowIdleBehavior()
  {
    TimeToUnload = TimeSpan.FromSeconds(0)
  };
  workflowServiceHost.Description.Behaviors.Add(
        workflowIdleBehavior);

  workflowServiceHost.Description.
        Behaviors.Find<ServiceDebugBehavior>().
        IncludeExceptionDetailInFaults = true;

  workflowServiceHost.Open();

  Console.WriteLine("WorkflowServiceHost started.");
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
  if (ex.InnerException != null)
  {
    Console.WriteLine(ex.InnerException.Message);
  }
}
Console.ReadKey();
```

Файл CreatingtheBusinessProcess.zip

Тестирование корректности предоставления метаданных

Запустите приложение `HolidayRequestProcessHost` извне Visual Studio (это обязательно следует делать от имени администратора). Откройте браузер и перейдите по адресу `http://localhost:9874/HolidayRequestProcess`. Если все сделано верно, на экране должна появиться страница со ссылкой на `http://localhost:9874/HolidayRequestProcess?wsdl` (рис. 13.42). Разработайте приложение `EmployeeHolidayRequestApplication`.

Во время работы `HolidayRequestProcessHost` вне Visual Studio откройте приложение `EmployeeHolidayRequestApplication` и добавьте ссылку на службу `HolidayRequestProcess`. В окне Add Service Reference введите адрес `http://localhost:9874/HolidayRequestProcess?wsdl` и установите пространство имен `HolidayRequestService` (рис. 13.43).

Добавьте в форму кнопку со следующим кодом.

```
HolidayRequestService.ServiceClient client;
client = new HolidayRequestService.ServiceClient();
HolidayRequestService.HolidayRequestDataContract_Output
            output;
output = client.RequestHoliday(
new HolidayRequestService.HolidayRequestDataContract_Input(
```

```
{
    EmployeeID = 101,
    HolidayEndDate = System.DateTime.Now,
    HolidayStartDate = System.DateTime.Now
});
MessageBox.Show(output.ReferenceID.ToString());
```


```
<?xml version="1.0" encoding="utf-8" ?>
<wsdl:definitions name="HolidayRequestProcessDefinition" targetNamespace="http://tempuri.org/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:tns="http://tempuri.org/" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy" xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
  xmlns:wsa10="http://www.w3.org/2005/08/addressing" xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
 - <wsdl:types>
  - <xsd:schema targetNamespace="http://tempuri.org/Imports">
      <xsd:import schemaLocation="http://localhost:9874/HolidayRequestProcess?xsd=xsd0" namespace="http://tempuri.org/" />
      <xsd:import schemaLocation="http://localhost:9874/HolidayRequestProcess?xsd=xsd1" namespace="http://schemas.microsoft.com/2003/10/Serialization/" />
      <xsd:import schemaLocation="http://localhost:9874/HolidayRequestProcess?xsd=xsd2" namespace="http://schemas.datacontract.org/2004/07/HolidayRequestDataContracts" />
    </xsd:schema>
  </wsdl:types>
 - <wsdl:message name="IService_RequestHoliday_InputMessage">
    <wsdl:part name="parameters" element="tns:RequestHoliday" />
  </wsdl:message>
 - <wsdl:message name="IService_RequestHoliday_OutputMessage">
    <wsdl:part name="parameters" element="tns:RequestHolidayResponse" />
  </wsdl:message>
 - <wsdl:message name="IService_ApproveRequest_InputMessage">
    <wsdl:part name="parameters" element="tns:ApproveRequest" />
  </wsdl:message>
 - <wsdl:message name="IService_ApproveRequest_OutputMessage">
    <wsdl:part name="parameters" element="tns:ApproveRequestResponse" />
  </wsdl:message>
 - <wsdl:portType name="IService">
  - <wsdl:operation name="RequestHoliday">
      <wsdl:input wsaw:Action="http://tempuri.org/IService/RequestHoliday" message="tns:IService_RequestHoliday_InputMessage" />
      <wsdl:output wsaw:Action="http://tempuri.org/IService/RequestHolidayResponse" message="tns:IService_RequestHoliday_OutputMessage" />
    </wsdl:operation>
  - <wsdl:operation name="ApproveRequest">
      <wsdl:input wsaw:Action="http://tempuri.org/IService/ApproveRequest" message="tns:IService_ApproveRequest_InputMessage" />
      <wsdl:output wsaw:Action="http://tempuri.org/IService/ApproveRequestResponse" message="tns:IService_ApproveRequest_OutputMessage" />
    </wsdl:operation>
  </wsdl:portType>
 - <wsdl:binding name="BasicHttpBinding_IService" type="tns:IService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  - <wsdl:operation name="RequestHoliday">
      <soap:operation soapAction="http://tempuri.org/IService/RequestHoliday" style="document" />
    - <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    - <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  - <wsdl:operation name="ApproveRequest">
      <soap:operation soapAction="http://tempuri.org/IService/ApproveRequest" style="document" />
    - <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    - <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
 - <wsdl:service name="HolidayRequestProcessDefinition">
  - <wsdl:port name="BasicHttpBinding_IService" binding="tns:BasicHttpBinding_IService">
      <soap:address location="http://localhost:9874/HolidayRequestProcess" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


Рис. 13.42. WSDL-код `HolidayRequestProcess`

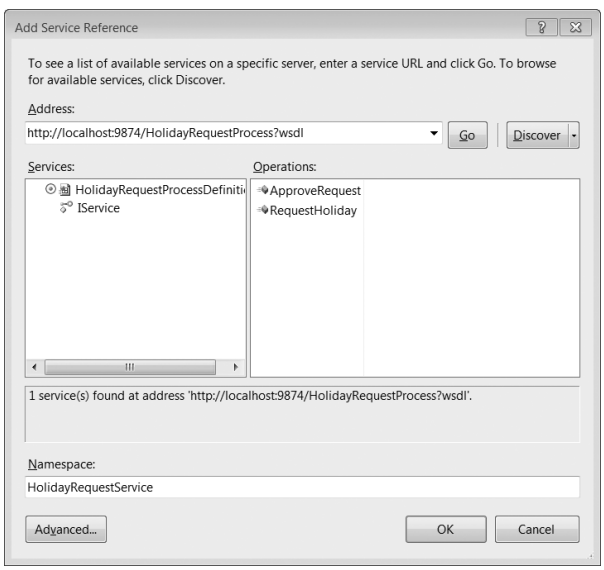


Рис. 13.43. Добавление ссылки на службу `HolidayRequestService`

Это не окончательный код, а всего лишь клиент для тестирования. В реальной ситуации приложение EmployeeHolidayRequestApplication имело бы куда более сложный интерфейс. Но это приложение позволяет нам протестировать рабочий процесс. Данный код просто вызывает метод прокси RequestHoliday, который запускает рабочий процесс, разработанный нами и размещенный с помощью workflowservicehost в HolidayRequestProcessHost. Результат вызова, полученный с идентификатором EmployeeID, выводится в окне сообщения.

Разработка ManagersHolidayRequestApprovalApplication

Это тестовое приложение, с помощью которого менеджер может утвердить или отклонить заявление об отпуске. Опять-таки это не более чем тестовое приложение, и в реальной ситуации оно должно иметь более сложный пользовательский интерфейс.

Добавьте ссылку на службу HolidayRequestProcess, как и для приложения для сотрудника. В диалоговом окне Add Service Reference введите в качестве адреса http://localhost:9874/HolidayRequestProcess и HolidayRequestService в качестве пространства имен для службы.

Добавьте в форму кнопку и текстовое поле ввода (которое требуется для referenceID). Добавьте следующий код для кнопки.

```
HolidayRequestService.ServiceClient client;
client = new HolidayRequestService.ServiceClient();
HolidayRequestService.HolidayApprovalDataContract_Input
    input;
input = new
    HolidayRequestService.HolidayApprovalDataContract_Input();
input.ManagerID = 1;
input.ReferenceID = int.Parse(textBox2.Text);

HolidayRequestService.HolidayApprovalDataContract_Output
    output;
output = client.ApproveRequest(input);

MessageBox.Show(output.EmployeeID.ToString());
```

Создание SqlWorkflowInstanceStore

Нам надо создать базу данных SQL для хранения состояния процесса. Эта база данных может работать под управлением SQL Express или SQL Server и должна включать соответствующие таблицы и хранимые процедуры, чтобы приложение workflowservicehost могло сохранять и получать состояние процесса. Имя базы данных должно быть указано в строке подключения connectionstring, передаваемой SqlWorkflowInstanceStoreBehavior и добавленной в WorkflowServiceHost.

Для создания необходимых таблиц и хранимых процедур следует запустить два сценария, которые можно найти в каталоге C:\Windows\Microsoft.NET\Framework\v4.0.30128\SQL\en, — SqlWorkflowInstanceStoreLogic.sql и SqlWorkflowInstanceStoreSchema.sql.

14

Хостинг

В ЭТОЙ ГЛАВЕ...

- Обзор возможностей хостинга
- Хостинг на сервере IIS\WAS
- Управление конечными точками и их отслеживание с помощью набора Windows Server AppFabric
- Перенаправление служб
- Хостинг на основе "облаков"

Платформа WCF прекрасно подходит для разработки служб. Но как сделать службу доступной всему миру после ее разработки? Чтобы запустить службу, необходим процесс хоста. Процесс хоста создает различные параметры конфигурации, обеспечивающие выполнение службы. Он готовит среду, создает конечные точки, запускает процессы прослушивания и управляет жизненным циклом службы.

Поскольку роль процесса хоста очень важна, он должен быть тщательно выбран. Есть две основные возможности хостинга службы.

- Локальный хостинг
  - Консольное приложение.
  - Приложение Windows.
  - Служба Windows.
- Сервер IIS
  - Сервер IIS 6 только для протокола HTTP.
  - Служба активизации Windows (WAS).
  - Набор Windows Server AppFabric.

Выбор локального хостинга позволяет размещать службы непосредственно в вашем приложении. Если хотите позволить приложению Windows получить сообщения или разрешить связь между разными процессами, используйте эту возможность. Это сделает ваше приложение совершенно независимым от конфигурации машины. Вы не обязаны устанавливать программное обеспечение стороннего производителя для хостинга своей службы. Поэтому вам придется управлять продолжительностью существования службы. Например, если процесс хоста неожиданно остановится, ваша служба окажется недоступной.

Выбор сервера IIS — надежный и эффективный способ хостинга. Это наилучший выбор, если вы работаете в распределенной среде и ваша служба разворачивается на выделенных серверах. Сервер IIS самостоятельно управляет сбором "мусора" приложения, его доступностью, надежностью и управляемостью.

Служба требует запуска процесса и не менее одного домена приложения. Домен приложения в среде .NET Framework представляет верховный уровень изоляции, когда выполняется управляемый код. Как показано на рис. 14.1, одиночный процесс Windows способен выполняться в нескольких доменах приложения.

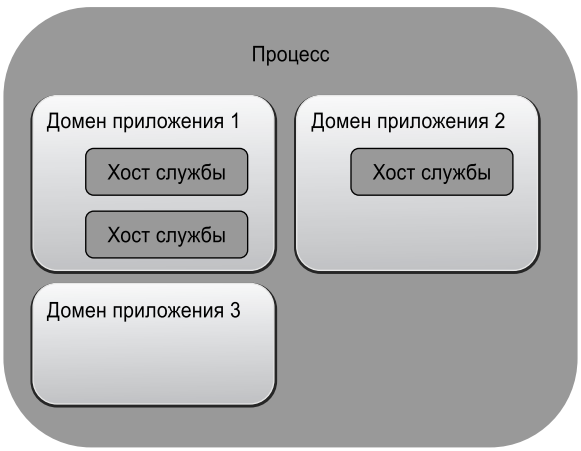


Рис. 14.1. Одиночный процесс Windows

Локальный хостинг приложений

Как упоминалось в предыдущем разделе, вы можете содержать свою службу в виде любого приложения .NET. Типичный пример — приложение, которое выполняется в фоновом режиме и фиксирует запросы; оно должно помещать сообщения в очередь и обработать их позже. В этом случае фоновое приложение может реализовать и содержать службу, открывать канал и прослушивать входящие сообщения.

Решение локального хостинга должно выполняться в процессе Windows. Оно унаследовало бы от процесса хостинга все параметры, такие как безопасность. Управление средой хостинга полностью на вашей ответственности. Вы должны контролировать жизненный цикл хоста (и службы). Решение локального хостинга полезно, когда необходимо обеспечить связь с различными компонентами вашего приложения, как правило, в локальной среде, где количество клиентов и транзакций весьма ограничено.

Реализация локального хостинга является самым простым способом хостинга ваших служб. Рассмотрим, как это сделать.

Классы ServiceHost и ServiceHostBase

Для хостинга службы в приложении .NET необходимо использовать специфический класс, ServiceHost, который позволяет установить среду выполнения служб. Как представлено в листинге 14.1, параметры службы можно установить программно, с помощью файла конфигурации или за счет комбинации обоих подходов.

Листинг 14.1. Использование класса ServiceHost


```
using Wrox.CarRentalService.Implementations.Europe;
...
ServiceHost carRentalHost = new ServiceHost(typeof(CarRentalService));
carRentalHost.Open();
```

Код листинга 14.1 создает экземпляр класса ServiceHost, связанного с реализацией службы Wrox.CarRentalService.Implementations.Europe.CarRentalService. Важнейшей особенностью является то, что каждый экземпляр класса ServiceHost способен содержать одновременно службу только одного типа. Обратите внимание на то, что к хосту не добавляется никаких объектов класса ServiceEndpoint. В данном случае при вызове метода Open хост службы пытается найти параметры в текущем файле конфигурации (Web.config или App.config).

Начиная с версии 4.0 платформы WCF, если раздел конфигурации не найден, класс ServiceHost может использовать стандартные параметры конечных точек (см. главу 3). В случае локального хостинга необходимо предоставить список базовых адресов конечных точек, которые будет использовать хост службы, а затем явно вызвать метод AddDefaultEndpoints(), чтобы создать конечные точки. Полный код представлен в листинге 14.2.

Листинг 14.2. Использование класса ServiceHost для добавления стандартных конечных точек


```
...
Uri[] baseAddresses = new Uri[]
{
    new Uri("http://localhost:10101/CarRentalService"),
    new Uri("net.tcp://localhost:10102/CarRentalService")
};

ServiceHost host = new ServiceHost(typeof(CarRentalService), baseAddresses);
host.AddDefaultEndpoints();
try
{
    host.Open();

    Console.WriteLine("The car rental service is up and listening on the
    endpoints:");
    foreach (var endpoint in host.Description.Endpoints)
    {
        Console.WriteLine("\t" + endpoint.Address.Uri.ToString());
    }
    Console.ReadLine();
```

```
    host.Close();
}
catch (CommunicationException ex)
{
host.Abort();
}
catch (TimeoutException ex)
{
    host.Abort();
}
catch (Exception ex)
{
    host.Abort();
    throw;
}
```

Листинг 14.2 демонстрирует, как можно создать экземпляр класса `ServiceHost` и открыть каналы, чтобы запустить прослушивание и получать сообщения. Обратите внимание на применение блока `try...catch` и метода `Close()`. Это наилучший способ управления состоянием объекта класса `ServiceHost`. Вы должны вызвать метод `Close()` только при успешном завершении потока приложения. В противном случае, когда передается исключение, необходимо вызывать метод `Abort()`, чтобы гарантировать немедленное завершение и закрытие каждого канала. Все подключенные клиенты получают исключение `CommunicationException`. Хотя класс `ServiceHost` реализует интерфейс `IDisposable`, не используйте блок `using`, поскольку это приводит к вызову метода `Dispose`, а затем, внутренне, к вызову метода `Close()`, даже при передаче исключения. Это приводит к непредвиденным исключениям, так как метод `Close()` ожидает для завершения текущей работы, что объект класса `ServiceHost` находится в корректном состоянии.

Вывод этого кода представлен на рис. 14.2.

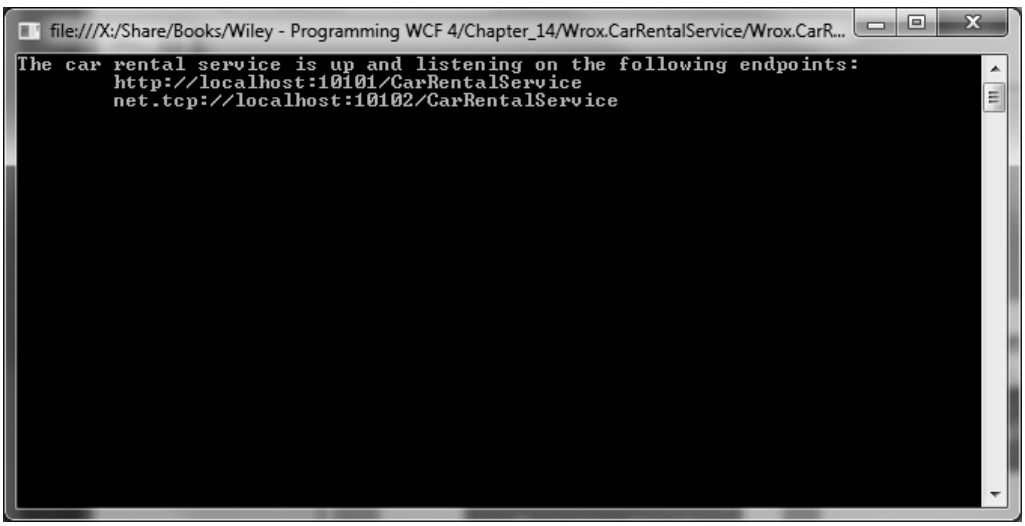


Рис. 14.2. Вывод кода

Хотя для каждой реализации службы можно использовать только один объект класса `ServiceHost`, вы вполне можете предоставить несколько интерфейсов, а затем несколько конечных точек. В том же самом процессе можете создать несколько экземпляров класса `ServiceHost`, чтобы содержать службы разных типов или того же типа.

Вызов методов `Open` и `Close` позволяет службе прослушивать на предопределенном адресе и готовит ее к получению сообщений. Как показано в листинге 14.3, можно также использовать асинхронные методы `BeginOpen/EndOpen` и `BeginClose/EndClose`.

Листинг 14.3. Асинхронные методы класса `ServiceHost`


```
using Wrox.CarRentalService.Implementations.Europe;
...

Uri[] baseAddresses = new Uri[]
{
    new Uri("http://localhost:10101/CarRentalService"),
    new Uri("net.tcp://localhost:10102/CarRentalService")
};

ServiceHost host = new ServiceHost(typeof(CarRentalService), baseAddresses);
host.AddDefaultEndpoints();
IAsyncResult result = host.BeginOpen(
    new AsyncCallback(ServiceHostOpenCallback), null);
```

Вызов метода `ServiceHostOpenCallback` происходит после завершения выполнения метода `BeginOpen`. В этом методе, соответствующем делегату `System.AsyncCallback`, можете разместить действие, выполняемое на этом этапе (листинг 14.4).

Листинг 14.4. Обработчик обратного вызова класса `ServiceHost`


```
ServiceHost cachedHost;
public void ServiceHostOpenCallback(IAsyncResult result)
{
    ...
}
```

Реализация специального хоста службы

Нередки ситуации, когда может потребоваться многократно использовать тот же хост службы с теми же параметрами, но в разных местах. Например, вы могли бы создать хост службы для программной настройки службы. В таком случае дополните класс `ServiceHost` и напишите специальную реализацию.

Класс `ServiceHost` происходит от абстрактного класса `ServiceHostBase`. Можете наследовать класс `ServiceHostBase` или `ServiceHost`, в который добавлено только четыре перегруженных метода `AddServiceEndpoint` и управление созданием экземпляра одиночной службы.

У платформы .NET Framework также есть реализации специальных хостов.

- Класс `System.ServiceModel.Web.WebServiceHost`, доступный в сборке `System.ServiceModel.Web`, происходит от класса `ServiceHost` и заставляет привязку `WebHttpBinding` и режим `WebHttpBehavior` автоматически создавать среду для выполнения служб на основе архитектуры REST.

- Класс `System.ServiceModel.WorkflowServiceHost`, доступный в сборке `System.WorkflowServices`, происходит от класса `ServiceHostBase`. Он инициализирует контекст рабочего потока и добавляет в экземпляр хоста режим `WorkflowRuntimeBehavior`.

Обе реализации хоста упрощают создание экземпляра хоста. То же самое вы можете сделать с помощью класса `ServiceHost`, но в данном случае вам придется писать каждый раз больше кода.

Как демонстрирует листинг 14.5, наследование от класса `ServiceHost` необходимо только при явной реализации абстрактного метода `ApplyConfiguration`, чтобы применить специфическую логику хоста.

Листинг 14.5. Создание специального хоста службы


```
public class CustomServiceHost : ServiceHost
{
    public CustomServiceHost(Type serviceType, params Uri[] baseAddresses)
        : base(serviceType, baseAddresses)
    {
    }

    protected override void ApplyConfiguration()
    {
        base.ApplyConfiguration();
        // здесь специальная логика
    }
}
```

Вызов метода `ApplyConfiguration` осуществляется из метода `InitializeDescription` класса `ServiceHostBase`. Можете переопределить его. Как правило, метод `InitializeDescription` вызывается из конструктора класса. Если вы не вызываете конструктор базового класса, придется вызвать метод `InitializeDescription` непосредственно; в противном случае ваша специальная реализация не сможет работать правильно.

Использование метода `ApplyConfiguration` позволяет изменять параметры только после того, как они прочитаны из файла конфигурации. Если хотите изменить логику загрузки конфигурации, не обращайтесь к базовой реализации, а напишите собственную специальную логику.

Еще один трюк подразумевает применение специальной логики хоста по возможности позднее, после применения всех конфигурационных и программных параметров, как представлено в листинге 14.6. Так ваши параметры не будут изменены после создания экземпляра хоста. Вы можете сделать это при событии `Opening`, прежде чем процесс `Open` будет завершен.

Листинг 14.6. Программная конфигурация специального хоста службы


```
public class CustomServiceHost : ServiceHost
{
    public CustomServiceHost(Type serviceType, params Uri[] baseAddresses)
        : base(serviceType, baseAddresses)
```

```
    {
        this.Opening += new EventHandler(CustomServiceHost_Opening);
    }

    void CustomServiceHost_Opening(object sender, EventArgs e)
    {
        MyCustomBehavior behavior =
            this.Description.Behaviors.Find<MyCustomBehavior>();
        if (behavior == null)
        {
            this.Description.Behaviors.Add(new MyCustomBehavior());
        }
    }
}
```

Наследование от класса `ServiceHost` является самым простым способом создания специальной реализации хоста, и это правильно в большинстве сценариев. Если ваш хост отличается от "классической" программной модели WCF, такой как `WorkflowServiceHost`, рассмотрите возможность наследования от класса `ServiceHostBase`. Это связано с тем, что у класса `ServiceHostBase` есть абстрактный метод `CreateDescription`, позволяющий создавать описание служб и определять реализуемые контракты. Класс `ServiceHost` использует этот метод для создания описания службы на основании программной модели WCF и применения ее атрибутов, таких как `ServiceContractAttribute`, `OperationContractAttribute` и `ServiceBehaviorAttribute`. Если хотите создать собственную модель программирования для построения служб, то должны наследовать от класса `ServiceHostBase`. Для этого применяется класс `WorkflowServiceHost`, но в другом случае наследование от класса `ServiceHost` — правильный выбор.

Хостинг на сервере IIS

В сложных случаях, когда ваша служба предоставляется в распределенной среде, необходимо использовать очень надежный, эффективный и управляемый хостинг. Действительно, чтобы обслуживать различные клиентские запросы, ваши службы должны быть надежными и выполняться всегда. В данном случае подойдет решение, предоставляемое веб-сервером информационных служб Интернета (Internet Information Services — IIS).

Хостинг на сервере IIS известен также как управляемый хостинг и обеспечивает ряд важных преимуществ.

- Процесс сбора "мусора". Частично решает проблему утечки памяти.
- Концепция пула приложения. Обеспечивает изоляцию приложений.
- Изолированный рабочий процесс. Для каждого пула приложения существует отдельный рабочий процесс, чтобы обеспечить средства масштабируемости, производительности и изоляцию при отказе.
- Активизация на основании сообщений. Позволяет создавать экземпляр и открывать хост, когда поступает новое входящее сообщение. Хост подчинен пулу экземпляров и многократно используется, когда поступает новый запрос.
- Мониторинг процесса. Гарантирует доступность процесса обработки входящих запросов. Если ни один рабочий процесс не доступен, инициализируется и запускается новый.

Служба хостинга платформы WCF на сервере IIS построена поверх конвейера HTTP ASP.NET. Это значит, что данный вид хоста поддерживает только транспортный протокол HTTP/HTTPS. У конвейера HTTP ASP.NET есть концепция обработчиков HTTP и модуля HTTP. Модуль HTTP — это специальный класс, который вы можете использовать для перехвата и манипуляции поступающими и исходящими сообщениями, обработчик HTTP — это класс, отвечающий за обработку конкретного сообщения. Обработчик HTTP активизируется при сопоставлении расширения с классом, который реализует интерфейс `IHttpHandler`. В случае платформы WCF для идентификации службы используется расширение файла `.svc`.

В среде разработки Visual Studio можете выбрать новый проект и создать приложение веб-сайта с помощью шаблона WCF Service, как представлено в листинге 14.7. Созданный проект содержит файл конфигурации, типовой контракт и реализацию службы, а также файл `.svc`, который ссылается на реализацию службы.

Листинг 14.7. Файл `.svc`, созданный шаблоном


```
<%@ ServiceHost Language="C#" Debug="true"
Service="Service" CodeBehind="~/App_Code/Service.cs" %>
```

Если открыть файл `.svc` в браузере, то можно увидеть справочную страницу, представленную на рис. 14.3. На ней есть ссылка, указывающая на документ WSDL, представленный на рис. 14.4, его можно просмотреть, если в настройке по умолчанию разрешено отображение метаданных.

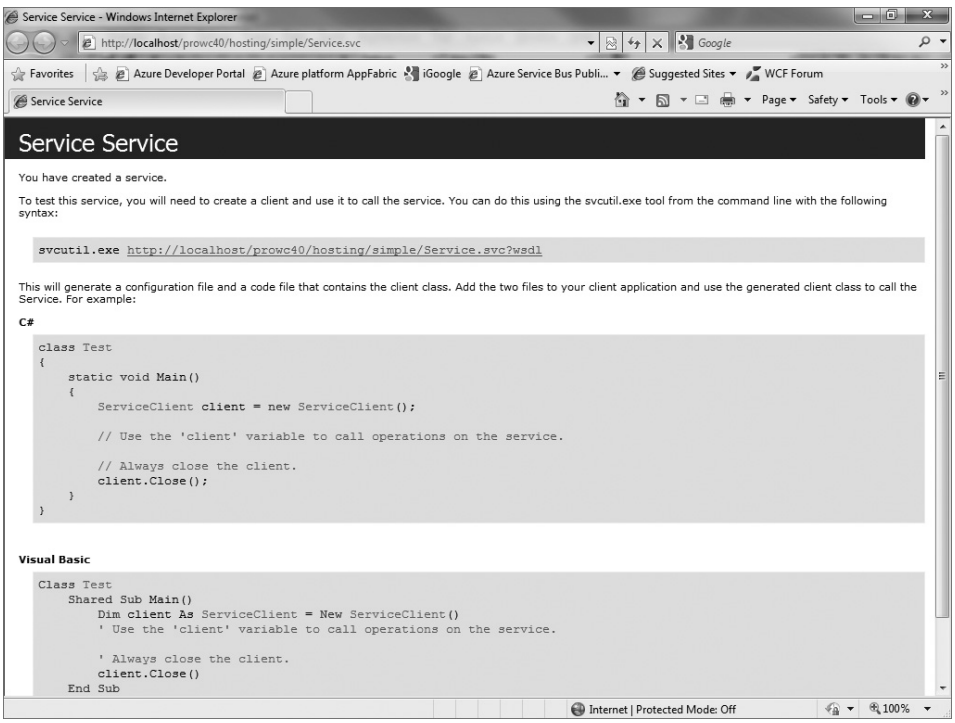
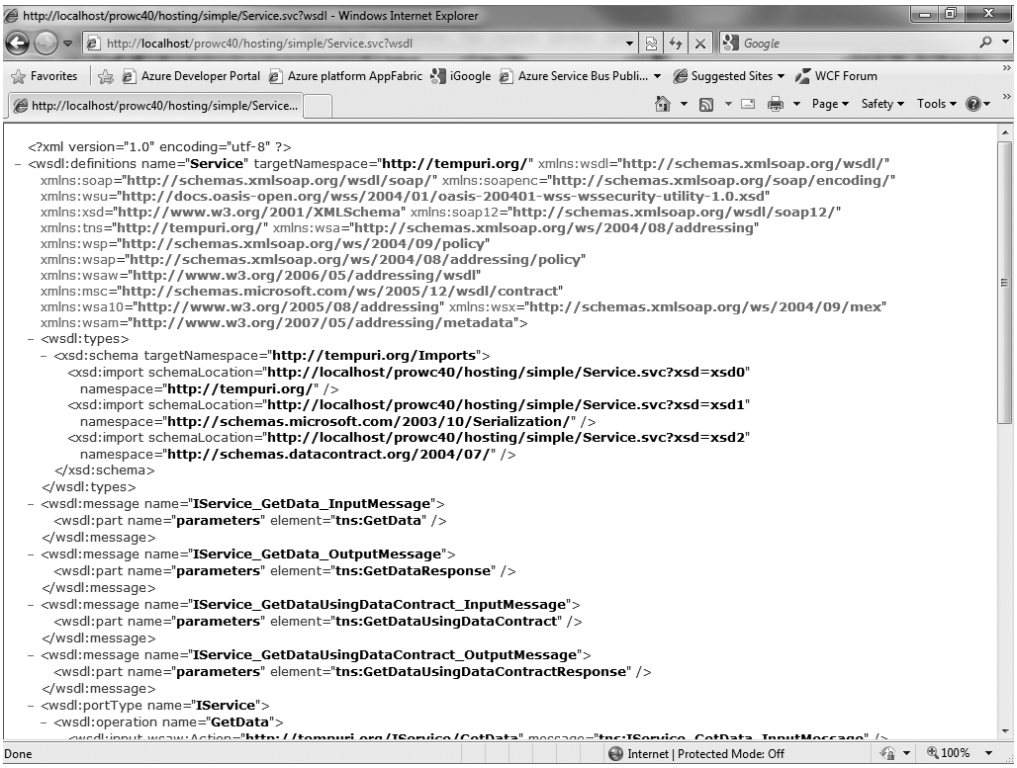


Рис. 14.3. Справочная страница




```
<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions name="Service" targetNamespace="http://tempuri.org/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://tempuri.org/" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
    xmlns:wsa10="http://www.w3.org/2005/08/addressing" xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
  - <wsdl:types>
    - <xsd:schema targetNamespace="http://tempuri.org/Imports">
        <xsd:import schemaLocation="http://localhost/prowc40/hosting/simple/Service.svc?xsd=xsd0"
          namespace="http://tempuri.org/" />
        <xsd:import schemaLocation="http://localhost/prowc40/hosting/simple/Service.svc?xsd=xsd1"
          namespace="http://schemas.microsoft.com/2003/10/Serialization/" />
        <xsd:import schemaLocation="http://localhost/prowc40/hosting/simple/Service.svc?xsd=xsd2"
          namespace="http://schemas.datacontract.org/2004/07/" />
      </xsd:schema>
    </wsdl:types>
  - <wsdl:message name="IService_GetData_InputMessage">
      <wsdl:part name="parameters" element="tns:GetData" />
    </wsdl:message>
  - <wsdl:message name="IService_GetData_OutputMessage">
      <wsdl:part name="parameters" element="tns:GetDataResponse" />
    </wsdl:message>
  - <wsdl:message name="IService_GetDataUsingDataContract_InputMessage">
      <wsdl:part name="parameters" element="tns:GetDataUsingDataContract" />
    </wsdl:message>
  - <wsdl:message name="IService_GetDataUsingDataContract_OutputMessage">
      <wsdl:part name="parameters" element="tns:GetDataUsingDataContractResponse" />
    </wsdl:message>
  - <wsdl:portType name="IService">
    - <wsdl:operation name="GetData">
```

Рис. 14.4. Документ WSDL

Класс `ServiceHost`, описанный в разделе локального хостинга, используется также при управляемом хостинге. Но на сей раз для успешного создания экземпляра хоста необходим другой компонент — `ServiceHostFactory`.

Классы `ServiceHostFactory` и `ServiceHostFactoryBase`

Подобно классам `ServiceHostBase` и `ServiceHost`, у платформы WCF есть классы `ServiceHostFactoryBase` и `ServiceHostFactory`. Эти два класса предоставляют методы фабрики для создания экземпляра хоста в управляемых средах, поддерживающих динамическую активизацию. Как вы уже видели, модель расширяемости WCF позволяет создавать собственную реализацию класса `ServiceHost`. Но в данном случае, отличном от случая локального хоста, управляемая среда, такая как сервер IIS, ничего не знает о вашем специальном хосте. В данном случае хост IIS должен использовать класс фабрики как точку входа, чтобы создать экземпляр хоста. Стандартной фабрикой, вызываемой сервером IIS, является класс `ServiceHostFactory`, но вы можете также создать собственный класс.

Метод `CreateServiceHost` — это самый важный метод, который необходимо переопределить. Его первый параметр — строковый параметр `constructorString`, который представляет нечто, используемое для получения корректной службы для хоста. В реализации класса `ServiceHostFactory` это должно быть полное имя типа службы (например, `CarRentalService.Implementations.Europe.CarRentalService`). Второй параметр — это список базовых адресов, унаследованный от хоста.

Чтобы использовать класс `ServiceHostFactory` (или `ServiceHostFactoryBase`), добавьте ссылку на сборку `System.ServiceModel.Activation`. Листинг 14.8 демонстрирует простую реализацию вашего класса `CustomServiceHost`.

Листинг 14.8. Создание специального класса `ServiceHostFactory`


```
public class CustomServiceHostFactory : ServiceHostFactory
{
    public override ServiceHostBase CreateServiceHost(
        string constructorString, Uri[] baseAddresses)
    {
        return new CustomServiceHost(Type.GetType(constructorString),
            baseAddresses);
    }

    protected override ServiceHost CreateServiceHost(
        Type serviceType, Uri[] baseAddresses)
    {
        return new CustomServiceHost(serviceType, baseAddresses);
    }
}
```

Логика фабрики должна остаться простой. Чтобы обеспечить повторное использование кода, необходимо переместить всю логику в класс специального хоста. Для этого имеет смысл расширять и использовать класс `ServiceHostFactoryBase`, когда вы наследуете класс `ServiceHostBase`, либо расширять и использовать класс `ServiceHostFactory`, когда вы наследуете класс `ServiceHost`.

Использование класса `CustomServiceHostFactory`

Как уже упоминалось, для хостинга службы на сервере IIS необходимо использовать в проекте веб-сайта файл `.svc`. Чтобы применить класс `CustomServiceHostFactory`, просто укажите это в свойстве `Factory` директивы `@ServiceHost` файла `.svc` (листинг 14.9).

Листинг 14.9. Объявление специальной фабрики


```
<%@ ServiceHost Language="C#" Service="Service" Factory="CustomServiceHostFactory" %>
```

Свойство `Factory` определяет реализацию класса `ServiceHostFactory`, экземпляр которой сервер IIS должен создать и вызвать. Значение `Service` свойства передается как параметр `constructorString` при вызове метода `CreateServiceHost`. В вашей специальной фабрике хоста значение параметра `constructorString` могло быть чем-то, используемым для восстановления службы, например, ключ для получения типа службы из совместно используемого хранилища, такого, как база данных или файл XML.

Службы IIS 7.0 и WAS, рассматриваемые подробно в следующем разделе, позволяют поддерживать несколько привязок сайтов. Список базовых адресов, передаваемых классу `ServiceHostFactory` сервером IIS, включает все имена хостов, которые сервер может принять. В большинстве случаев это означает, что такие адреса, как `http://localhost` и `http://machinename`, передаются классу `ServiceHost`, и происходит исключение с сообщением

`System.ArgumentException: This collection already contains an address with scheme Http. There can be at most one address per scheme in this collection.`
(`System.ArgumentException`: Эта коллекция уже содержит адрес со схемой HTTP. В этой коллекции может быть не больше одного адреса на схему.)

На платформе WCF 4.0 вы можете, наконец, отработать этот случай и позволить использование нескольких привязок. В файле конфигурации необходимо добавить параметр, представленный в листинге 14.10.

Листинг 14.10. Разрешение нескольких привязок


```
<system.serviceModel>
    <serviceHostingEnvironment multipleSiteBindingsEnabled="true" />
</system.serviceModel>
```

Активизация без файла SVC

Вы уже знаете, как можно организовать хостинг службы на сервере IIS\WAS, используя файл `.svc`. На основании расширения `.svc` среда выполнения, которая обрабатывает специфический запрос, соотносится с обработчиком `HttpHandler`. Это предопределено в базовом файле `Web.config`, расположенном в папке конфигураций платформы .NET Framework (листинг 14.11).

Листинг 14.11. Настройка обработчика `HttpHandler` для `.svc`


```
<httpHandlers>
    [...]
    <add path="*.svc" verb="*"
        type="System.ServiceModel.Activation.HttpHandler,
        System.ServiceModel.Activation, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" validate="False" />
    [...]
</httpHandlers>
```

В новой, четвертой версии платформы WCF объединение возможностей с сервером IIS позволяет создавать службы без необходимости определять файл `.svc`. Как? С помощью файла конфигурации, как представлено в листинге 14.12.

Листинг 14.12. Активизация без файла `.svc`


```
<system.serviceModel>
    [...]
    <serviceHostingEnvironment multipleSiteBindingsEnabled="true">
        <serviceActivations>
            <add relativeAddress="CarRentalService.svc"
service="Wrox.CarRentalService.Implementations.Europe.CarRentalService"/>
        </serviceActivations>
    </serviceHostingEnvironment>
    [...]
    <services>
        <service name="Wrox.CarRentalService.Implementations.Europe.CarRentalService">
            <endpoint address=""
```

```
                binding="basicHttpBinding"
                contract="Wrox.CarRentalService.Contracts.ICarRentalService" />
        </service>
    </services>
</system.serviceModel>
```

С этой конфигурацией можете открыть в браузере и просмотреть файл `CarRentalService.svc` и увидеть справочную страницу службы, как реальный файл `.svc`.

Службы активизации Windows

Платформа WCF позволяет создавать службы независимо от используемого транспортного протокола. Но с сервером IIS 6 можно использовать и предоставить службы только по протоколу HTTP с помощью конвейера ASP.NET Http Pipeline. Это большое ограничение, когда вы получаете преимущества архитектуры WCF, способной использовать широкий диапазон протоколов. Чтобы позволить применение полного списка протоколов, поддерживаемых платформой WCF, таких как net.tcp, net.pipe, net.msmq и msmq.formatname, используйте платформу, называемую службой активизации Windows (Windows Activation Services — WAS).

Служба WAS прослушивает на определенном протоколе и перенаправляет полученное сообщение рабочему процессу, чтобы обработать запрос. Архитектура WAS определяет следующие компоненты.

- Адаптер обработчика. Используется для получения сообщения по определенному протоколу и перенаправления его определенному рабочему процессу.
- Обработчик, специфичный для протокола. Выполняется в рабочем процессе и отвечает за обмен сообщениями с адаптером обработчика.

У сервера IIS создается предопределенный набор привязок, определяющих имя хоста, порт, IP-адрес и некоторую другую информацию, которую должна использовать каждая привязка (рис. 14.5). У каждого веб-сайта есть собственная конфигурация.

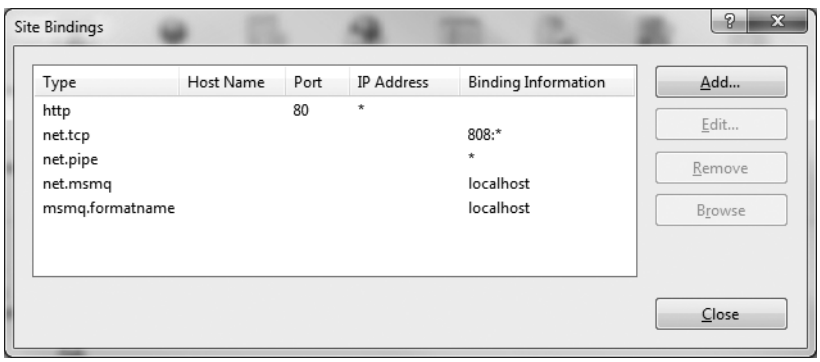


Рис. 14.5. Набор привязок сервера IIS

Установить применение определенного протокола очень просто. Во-первых, следует удостовериться, что привязка, относящаяся к необходимому протоколу, отличному от HTTP, настроена для веб-сайта (например, net.tcp на стандартном веб-сайте). Во-вторых, разрешите применение протокола в виртуальном каталоге, добавив, например, протокол net.tcp в список разрешенных протоколов в окне Advanced Settings (Дополнительные параметры), как показано на рис. 14.6.

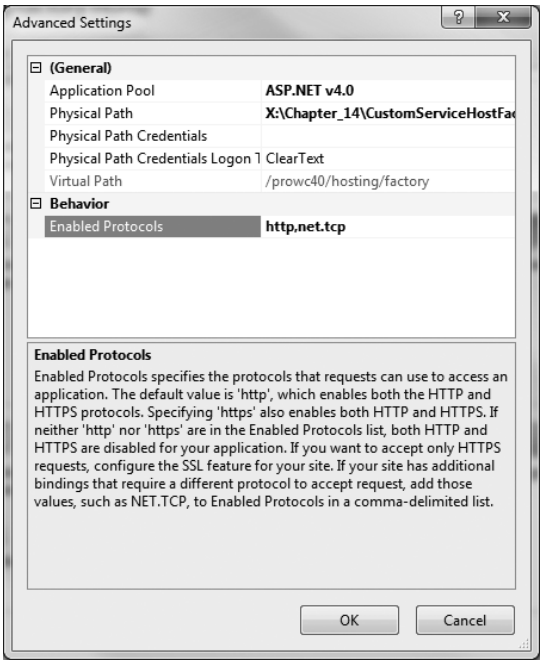


Рис. 14.6. Окно Advanced Settings

Можете также разрешить использование протокола, отличного от HTTP, с помощью инструмента командной строки `appcmd.exe`.

```
appcmd.exe set site "Default Web Site"
-+bindings.[protocol='net.tcp',bindingInformation='808:*']
```

Вы все еще можете использовать протокол HTTP, но теперь можно также использовать протокол net.tcp. И окно Advanced Settings, и инструмент `appcmd.exe` изменяют содержимое файла `applicationHost.config`, расположенного в папке `%windir%\system32\inetsrv\config` (листинг 14.13).

Листинг 14.13. Файл `applicationHost.config`


```
<sites>
    <site name="Default Web Site" id="1">
        <application path="/prowc40/hosting" applicationPool="DefaultAppPool">
            <virtualDirectory path="/"
                    physicalPath="C:\inetpub\wwwroot\prowc40\hosting" />
        </application>
        <bindings>
            <binding protocol="http" bindingInformation="*:80:" />
            <binding protocol="net.tcp" bindingInformation="808:*" />
            <binding protocol="net.pipe" bindingInformation="*" />
            <binding protocol="net.msmq" bindingInformation="localhost" />
            <binding protocol="msmq.formatname" bindingInformation="localhost" />
        </bindings>
    </site>
</sites>
```

Теперь протокол net.tcp стал частью основного списка адресов, предоставляемого с управляемого хоста сервера IIS. На платформе WCF 4.0 с появлением стандартных конечных точек необязательно определять конечную точку в файле конфигурации. Служба предоставляется с двумя адресами, по одному для каждого протокола: HTTP и net.tcp.

Если вы проверите созданный файл WSDL, то увидите правильный элемент порта net.tcp в разделе `wsdl:service`. Код XML в листинге 14.14 является частью созданного документа WSDL (расположение службы зависит от вашей специфической конфигурации).

Листинг 14.14. Раздел `wsdl:service` с конечной точкой net.tcp


```
<wsdl:service name="Service">
    <wsdl:port name="BasicHttpBinding_IService"
        binding="tns:BasicHttpBinding_IService">
        <soap:address location="http://localhost/Service.svc" />
    </wsdl:port>
    <wsdl:port name="NetTcpBinding_IService" binding="tns:NetTcpBinding_IService">
        <soap12:address location="net.tcp://localhost/Service.svc" />
        <wsa10:EndpointReference>
            <wsa10:Address>net.tcp://localhost/Service.svc</wsa10:Address>
        </wsa10:EndpointReference>
    </wsdl:port>
</wsdl:service>
```

Но если вы хотите изменить стандартные параметры конечной точки, то сделайте так, как показано в листинге 14.15.

Листинг 14.15. Установка специальной конфигурации net.tcp


```
<system.serviceModel>
    <services>
        <service name="Service">
            <endpoint address=""
                binding="netTcpBinding"
                bindingConfiguration="ServiceNetTcp"
                contract="IService">
            </endpoint>
        </service>
    </services>
    <bindings>
        <netTcpBinding>
            <binding name="ServiceNetTcp">
                <security mode="TransportWithMessageCredential"/>
            </binding>
        </netTcpBinding>
    </bindings>
</system.serviceModel>
```

Поскольку единственная конечная точка определена для протокола net.tcp, конечная точка для протокола HTTP больше не доступна. Служба прослушивает только адрес `net.tcp://localhost/Service.svc`.

Управление конечными точками и их отслеживание с помощью набора Windows AppFabric

Служба WAS — это только первый шаг создания из сервера IIS полного сервера приложений. Фактически у него отсутствуют некоторые из возможностей, в которых нуждаются реальные распределенные среды. Развертывание собственных служб, их настройка и проверка, мониторинг состояния и диагностика не учитываются сервером IIS/WAS.

Чтобы заполнить этот промежуток, корпорация Microsoft выпустила набор Windows Server AppFabric, уже известный как *Дублин*, — модификация для сервера WAS, который представляет ряд новых и очень важных дополнений для хостинга служб и рабочих потоков WCF. Набор AppFabric существенно упрощает способ контроля выполнения служб и рабочих потоков, а также их производительность.

Набор интегрированных технологий Windows Server AppFabric обеспечивает следующее.

- Развертывание. Упрощенный способ развертывания служб. Можно создать и развернуть свой пакет из среды разработки Visual Studio 2010, но можно также экспортировать и импортировать установочные пакеты из существующих приложений.
- Конфигурация. Интегрированный интерфейс с сервером IIS позволяет настраивать автозапуск приложений, производительность, подключение к базе данных для длительного рабочего потока, сертификат службы и некоторые другие возможности, которые администратор может изменить вручную, не модифицируя файл конфигурации.
- Мониторинг. Набор интегрированных технологий Windows Server AppFabric для платформы .NET 4 предоставляет новые инструментальные средства мониторинга для проверки состояния, исключений, а также количества завершенных и неудачных вызовов в сквозных сценариях.
- Хостинг. Надежный хостинг рабочего потока позволяет использовать базы данных с обеспечением совместного доступа на основе сервера SQL Server. Это позволяет возобновлять долго выполняющиеся рабочие потоки при остановке или перезапуске сервера.
- Распределенный кэш. Уже известный как Microsoft Velocity. Можете использовать эту возможность для кэширования элементов и их многократного использования на любых серверах вашей веб-фермы.

Мощный интерфейс, полностью интегрированный в диспетчер IIS Manager, существенно упрощает настройку приложений. С помощью набора интегрированных технологий Windows Server AppFabric администратор может модифицировать стандартные параметры служб без изменения и перекомпиляции кода или, что еще хуже, изменения файла `web.config` вручную. На рис. 14.7 показано, как архитектура Windows Server AppFabric полностью интегрируется в IIS.

Интегрированный набор инструментальных средств IIS, который прежде назывался Windows Application Server Extensions for .NET 4.0, позволяет осуществлять следующее.

- Настройка мониторинга, персистентности, доступности и автозапуска, а также производительность и безопасность на уровнях веб-приложений, веб-сайтов и служб.

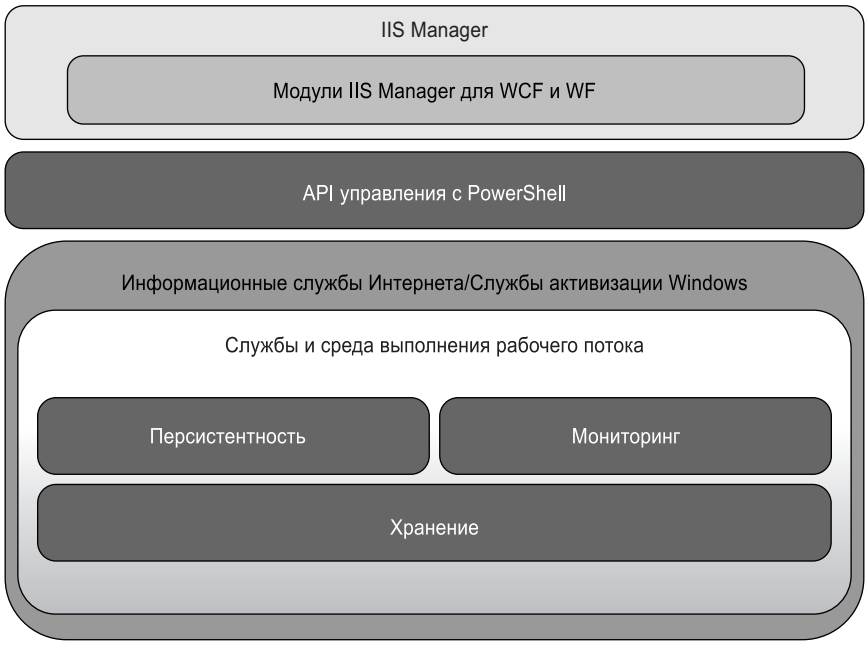


Рис. 14.7. Интеграция архитектуры Windows Server AppFabric

- Перечисление и настройка конечных точек определенной службы.
- Управление подключением, инициализация или обновление конфигурации базы данных мониторинга Monitoring Database Configuration.
- Управление подключением, инициализация или обновление конфигурации базы данных персистентности Persistence Database Configuration.
- Проверка в реальном времени корректности состояния служб и рабочих потоков за счет мониторинга завершенных, ошибочных или зависших обращений к службе, а также активных, ожидающих или приостановленных экземпляров рабочих потоков.

Кроме того, был выпущен набор команд PowerShell, что позволяет вам выполнять все те же действия, что и в предыдущих модификациях.

Перемещение между ними — очень простая вещь, поскольку модификации позволяют вам, например, получить список конечных точек для службы или службу выбранной конечной точки.

Кроме того, инструмент развертывания Web Deployment — еще одна модификация сервера IIS, устанавливаемая вместе со средой разработки Visual Studio 2010, но доступная также и для отдельной загрузки и обеспечивающая экспорт и импорт пакетов прикладных программ непосредственно из диспетчера IIS Manager. Затем можете экспортировать свое веб-приложение, включая параметры, и импортировать его на другой сервер вашей фермы серверов. Среда разработки Visual Studio 2010 позволяет также создать тот же установочный пакет из проводника решений Solution Explorer. В настоящее время платформе WCF недостает процесса развертывания, но инструмент развертывания Web Deployment предоставляет очень простой и мощный способ установки ваших служб.

Корпорация Microsoft выпустила также набор AppFabric для платформы Windows Azure, который облегчает пользователям разработку решений в "облаке", в то время как набор Windows Server AppFabric предназначен для "классических" сред. Набор AppFabric для платформы Windows Azure, в отличие от набора Windows Server AppFabric, предоставляет решение Service Bus для облегчения связи между службами и службой управления доступом Access Control Service (ACS), а также создания маркеров доступа для интегрированной авторизации. Набор AppFabric для платформы Windows Azure обсуждается далее в этой главе.

В следующих разделах вы увидите, как настроить и использовать инструментальные средства набора интегрированных технологий Windows Server AppFabric, чтобы контролировать службы WCF и WF.

Установка набора Windows Server AppFabric

Набор Windows Server AppFabric доступен как отдельный пакет установки, загружаемый с сайта Microsoft по адресу `http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx`. Набор Windows Server AppFabric выполняется только на операционных системах Windows Vista, Windows 7, Windows Server 2008 и Windows Server 2008 R2. Некоторые средства, такие как автозапуск, выполняются только на операционной системе Windows 7 и Windows Server 2008 R2. Сервер SQL Server не обязателен для установки набора Windows Server AppFabric, но некоторые средства, такие как мониторинг, персистентность или кэширование, не доступны, если он отсутствует. Можете также использовать сервер SQL Server 2008, но сервер SQL Server 2005 не поддерживается. Можно также использовать другие базы данных или хранилища, особенно для персистентности и кэша.

После завершения процесса установки запустите мастер конфигурации, чтобы создать и инициализировать базу данных мониторинга и базу данных персистентности, как показано на рис. 14.8.

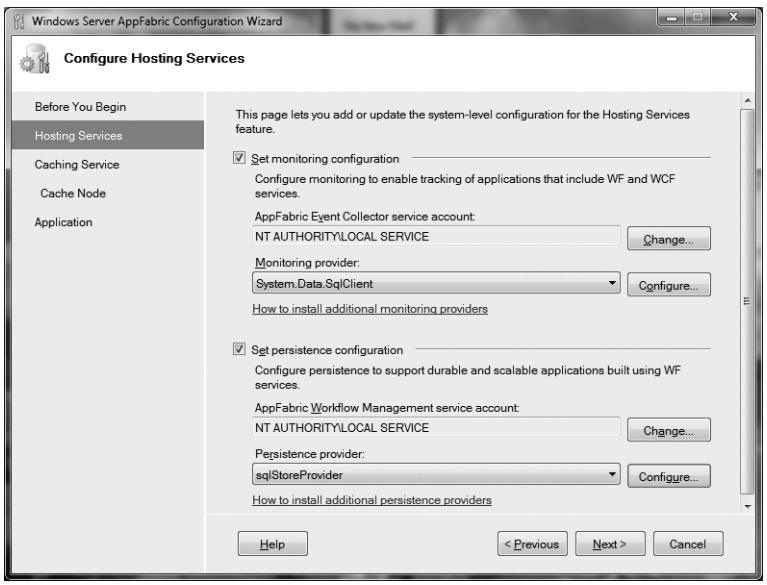


Рис. 14.8. Мастер конфигурации

Щелкните на кнопке Configure (Настроить), чтобы инициализировать или просто сослаться на уже созданную базу данных (рис. 14.9).

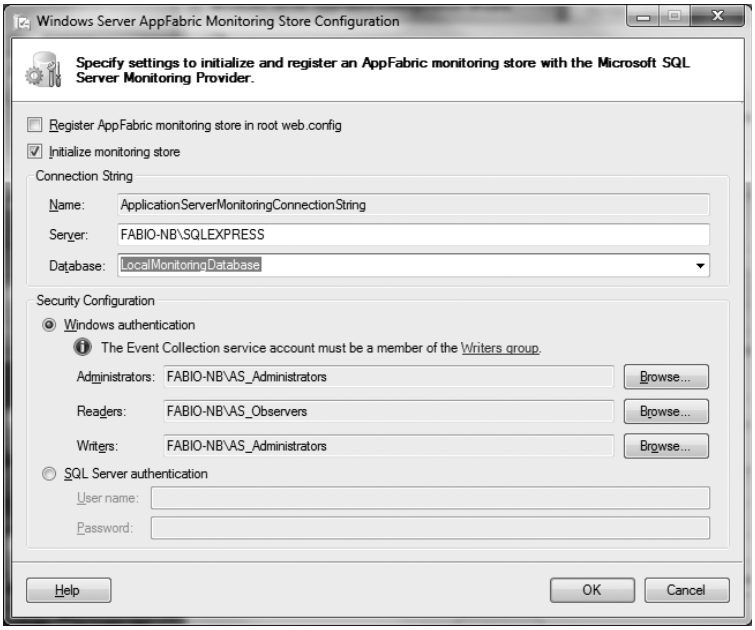


Рис. 14.9. Выбор базы данных

Если хотите настроить эти параметры позже, то запустите мастер конфигурации Windows Server AppFabric Configuration Wizard, выбрав в меню кнопки Пуск пункт All Programs⇨Windows Server AppFabric⇨Configure AppFabric (Все программы⇨Windows Server AppFabric⇨Configure AppFabric).

В конце мастера вы увидите новые разделы в интерфейсе IIS, которые позволяют получить доступ к средствам набора AppFabric.

Мониторинг служб с помощью набора AppFabric

Диагностика выполняющихся служб — не простое дело. Фактически единственный способ поиска исключения приложения или трассировки стека — это применение трассировки диагностики и регистрации сообщений платформы WCF. В большинстве таких случаев не так просто изменить файл конфигурации, поскольку там слишком много параметров среды. Вы должны быть в состоянии воспроизвести шаг, который приводит к отказу службы, получить файлы трассировки, а затем отключить трассировку, поскольку это существенно снижает производительность приложения, ведь происходит добавление строк в текстовый файл. Получив файлы результатов трассировки и регистрации, проанализируйте их с помощью средства просмотра трассировки служб Microsoft Service Trace Viewer (`SvcTraceView.exe`), поставляемого в комплекте Microsoft Windows SDK.

Одним из самых важных средств набора Windows Server AppFabric, обеспечивающих лучшее решение, является способность контролировать допустимость ваших служб с помощью простой интегрированной панели Dashboard, позволяющей быстро проверять службы и исследовать приложения WCF.

Вы не должны писать дополнительный код или изменять файлы конфигурации, поскольку среда выполнения WCF на платформе .NET 4.0 способна использовать высокоэффективную архитектуру трассировки событий Event Tracing for Windows (ETW). Архитектура ETW выходит за рамки рассмотрения этой книги. Более подробная информация по этой теме приведена в статье по адресу `http://msdn.microsoft.com/en-us/magazine/cc163437.aspx`.

Если вы пытаетесь найти ошибки во время работы приложения, необходимо проанализировать различные уровни сообщений (от простых исключений до сложных и подробных сообщений), а также некоторые этапы до и после того, как было передано исключение.

Набор AppFabric различает несколько уровней, из которых вы можете выбрать следующие.

- Off (Отключено). Монитор полностью отключен.
- Errors Only (Только ошибки). Данные отслеживаются только тогда, когда происходит исключение или событие предупреждения.
- Health Monitoring (Мониторинг состояния). Стандартный уровень, который позволяет отображать состояние служб WCF и WF с помощью инструментов набора AppFabric.
- End-to-End Monitoring (Сквозной мониторинг). Позволяет отслеживать поток сообщений, когда вызов затрагивает несколько служб. Если клиент вызывает службу, которая вызывает другую службу, чтобы закончить процесс обработки сообщения, весь поток восстанавливается и отслеживается.
- Troubleshooting (Диагностика). Самый подробный уровень, который следует применять для диагностики проблем со службами WCF и WF.

Еще одно существенное отличие от простой трассировки — это пропускная способность отслеживания потока сообщений в сквозных сценариях. В распределенных средах есть множество взаимодействующих служб. Одно сообщение может пройти через несколько служб, прежде чем оно будет обработано. При двухточечных связях клиент посылает сообщение службе; при сквозных связях клиент посылает сообщение службе, которая перенаправляет его другой службе, и т.д. Можете настроить уровень с помощью инструмента конфигурации WCF and WF Configuration, поставляемого в наборе AppFabric.

Мониторинг набора AppFabric обеспечивает корреляцию трассировки WCF, позволяя вам выяснить, что происходит с сообщением при передаче от клиента к конечной службе и назад. Если вы отрабатываете несколько серверов, как на веб-ферме, использование общей и совместно используемой базы данных SQL Server позволяет собрать коллекцию событий и выявить корреляцию сообщений по серверам.

Для запуска мониторинга не нужно изменять исходный код. Достаточно щелкнуть правой кнопкой мыши на узле веб-приложения на сервере IIS и выбрать пункт .NET WCF and WF, а затем щелкнуть на кнопке Configure (Настроить), чтобы открыть окно инструмента Configure WCF and WF for Application (Конфигурация WCF и WF для приложений). Как показано на рис. 14.10, инструмент конфигурации позволяет установить для вашего приложения службу WCF или WF уровень Health Monitoring (Мониторинг состояния).

Этот инструмент позволяет также запустить средства трассировки и регистрации сообщений за несколько простых шагов без использования редактора конфигурации служб WCF Service Configuration Editor, интерфейс которого труден для использования и зачастую менее понятен. Набор AppFabric существенно упрощает для системных администраторов процесс настройки.

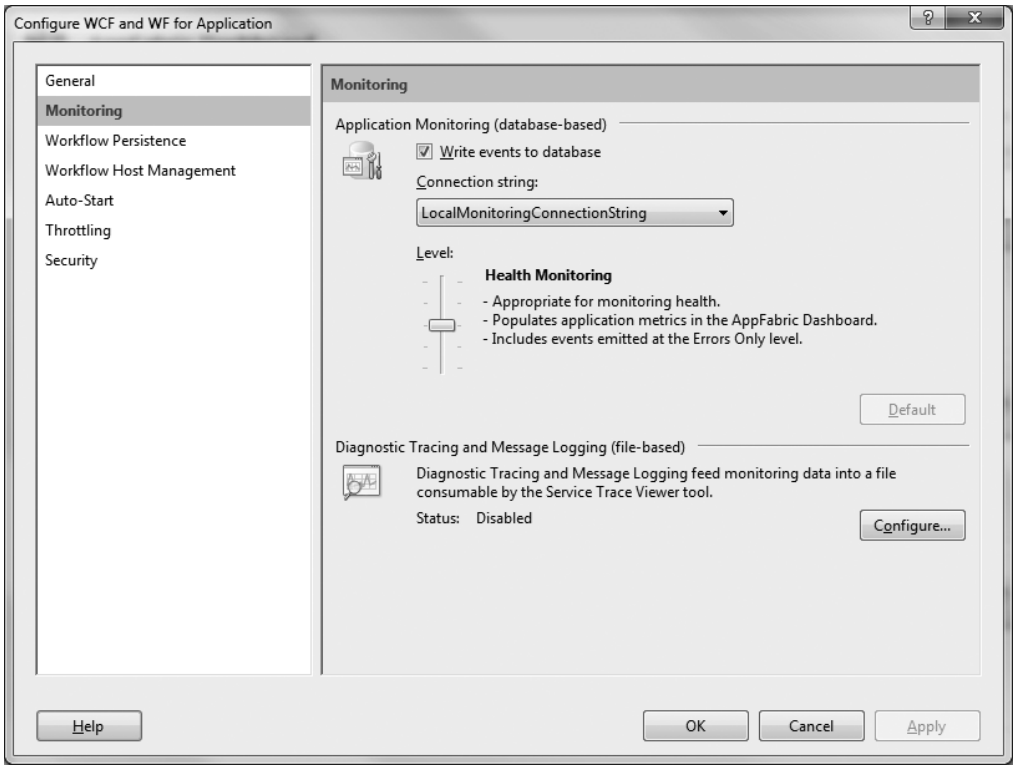


Рис. 14.10. Окно инструмента Configure WCF and WF for Application

Теперь, когда ваша служба выполняется на сервере IIS и установлен уровень Health Monitoring Level, можете контролировать свою службу с помощью панели Fabric Dashboard, представленной на рис. 14.11. Служба Windows — *AppFabric Event Collection Service* — собирает события сеанса ETW и записывает их в базу данных мониторинга.

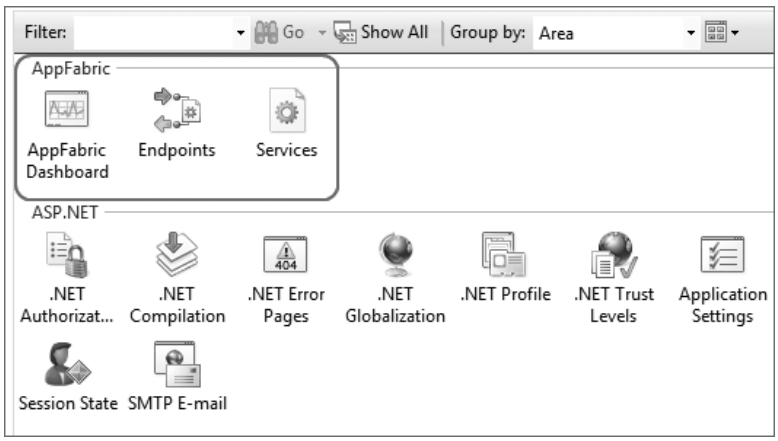


Рис. 14.11. Панель Fabric Dashboard

Панель Dashboard позволяет получать информацию о текущих выполняющихся запросах, о завершенных запросах, а также подробности о неудачных запросах. Как можно заметить на рис. 14.12, здесь есть три раздела.

- Completed Calls (Завершенные вызовы). Отображает список успешно завершенных вызовов, где каждый адрес находится в отдельном ряду.
- Service Exceptions (Исключения службы). Отображает список исключений, переданных службой во время выполнения.
- Failed or Faulted Calls (Ошибочные и сбойные вызовы). Создает ошибочные и сбойные вызовы при передаче исключения.

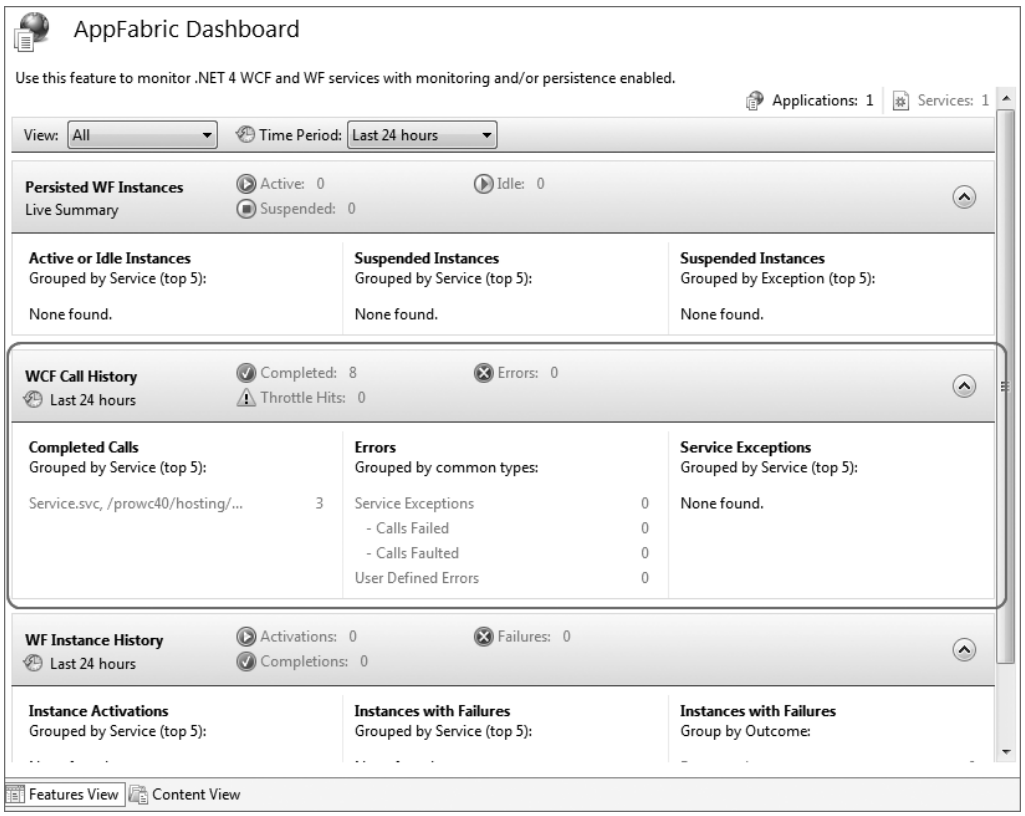


Рис. 14.12. Панель AppFabric Dashboard

После щелчка на отображаемом ряде создается запрос на создание данных отслеживания, а результат возвращается в виде списка собранных событий, отфильтрованных по переданным входным параметрам. Чтобы ограничить результат запроса, можете изменить входные параметры.

На рис. 14.13 приведен список собранных событий, отображаемых в реальном времени. Здесь также показан идентификатор события корреляции для сквозных сценариев `E2EActivityId`, если он активизирован.

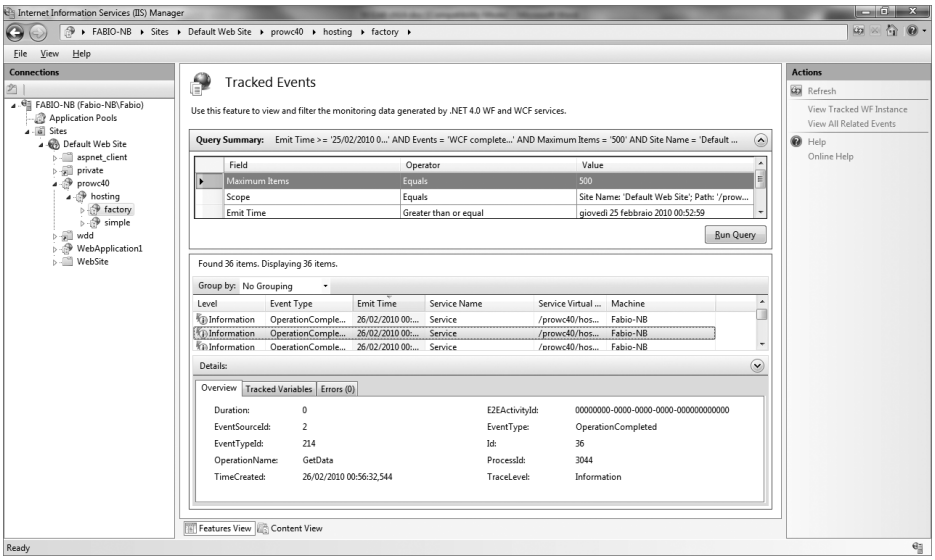


Рис. 14.13. Список собранных событий

Можете также отобразить список исключений сервера. На рис. 14.14 показан список исключений службы и связанные с ними подробности стека трассировки.

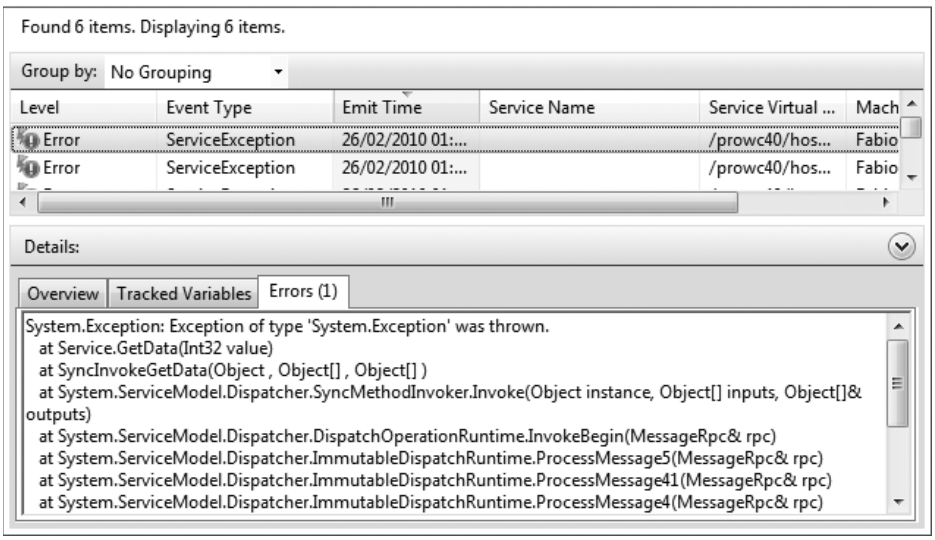


Рис. 14.14. Список исключений службы

Запуск просмотра событий

Иногда диагностика приложений, особенно служб, представляет собой очень трудное дело. Как уже упоминалось, до появления пакета обновлений WCF 3.5 Service Pack 1 вы могли запустить Diagnostic Trace and Message Logging, действительно хороший инструмент, но иногда трудный для понимания и поиска проблем.

На платформе WCF 4 с набором AppFabric можно извлечь пользу из применения инфраструктуры ETW. По умолчанию она отключена. Чтобы включить ее, откройте окно Event Viewer (Просмотр событий), разверните узлы Application and Services Logs (Журналы приложений и служб), Microsoft, Windows, а затем щелкните правой кнопкой мыши на узле Application Server-Applications. Выберите пункт View (Вид), установите флажок Show Analytic and Debug Logs (Показать журналы Analytic и Debug), если он еще не установлен. В узле Application Server-Applications выберите журнал Analytic, а затем щелкните на Enable Log Item (Включить журнал) в панели Actions (Действия).

Если вызвать теперь службу, в журнале Analytic отобразятся события, как показано на рис. 14.15.

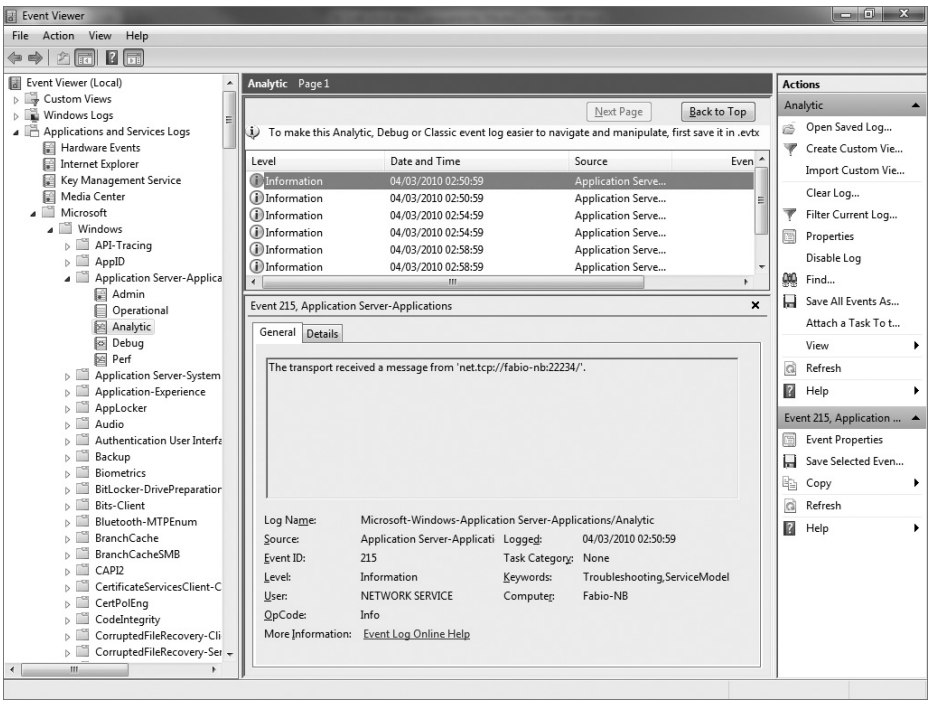


Рис. 14.15. События в журнале Analytic

Маршрутизация служб

При разработке распределенных приложений вам, возможно, понадобится создать интерфейсную службу, которая действует как мост между вашими внутренними службами, например, в защищенной внутренней сети. У платформы WCF 4.0 есть новая служба: `RoutingService`. Она позволяет создать службу внешнего интерфейса, которая получает запросы и перенаправляет сообщения на внутренний набор служб, развернутых в вашей закрытой сети, возможно, вне брандмауэра.

Здесь доступны следующие возможности.

- Маршрутизация с учетом содержимого.
- Протокол и мостовое соединение защиты.
- Обработка ошибок.

Служба RoutingService действует как посредник между клиентом и службой. Она получает сообщение от клиента и на основании примененных фильтров перенаправляет сообщение определенной службе.

Маршрутизация с учетом содержимого

Основная роль службы маршрутизации — перенаправление полученного от клиента сообщения на серверную службу, где оно будет обработано. Как она может выяснить правильную конечную службу, которая должна быть вызвана? Службе маршрутизации неизвестны сами серверные службы, но она знает сообщение и применяет к нему набор настроенных фильтров, чтобы индивидуализировать конечные службы, подлежащие вызову.

На платформе WCF 4.0 уже доступен и готов к использованию набор предопределенных фильтров. Вы можете применить фильтр на основании действия сообщения или запроса XPath. В табл. 14.1 описаны некоторые из типов фильтров и их применение.

Таблица 14.1. Фильтры службы маршрутизации

Тип фильтра	Описание
Action	Основан на действии сообщения, определенном в атрибуте filterData. Пример: <filter name="actionFilter" filterType="Action" filterData="http://wrox/CarRentalService/2009/10/RentalService/CalculatePrice"/>
And	Используется для указания названий двух других фильтров, результаты которых должны быть объединены и проверены на совпадение. Пример: <filter name="andFilter" filterType="And" filter1="firstFilter" filter2="secondFilter"/>
EndpointAddress	Основан на значении заголовка To сообщения. Пример: <filter name="endpointAddressFilter" filterType ="EndpointAddress" filterData=" http://localhost:10101/CarRentalService" />
EndpointAddressPrefix	Подобен фильтру EndpointAddressFilter. Соответствует началу значения заголовка To сообщения. Пример: <filter name="endpointAddressPrefixFilter" filterType="EndpointAddress" filterData="http://localhost:10101" />
EndpointName	Соответствует названию одной из конечных точек служб, предоставленных в службе маршрутизации. Пример: <filter name="endpointNameFilter" filterType= "EndpointName" filterData="routingEndpoint1" />
Custom	Позволяет определить специальный тип фильтра. Пример: <filter name="customFilter" filterType="Custom" customType="MatchVersionMessageFilter" filterData="Soap11" />
MatchAll	Соответствует всем входящим сообщениям. Пример: <filter name= "matchAllFilter" filterType="MatchAll" />
XPath	Применяет к входящим сообщениям выражение XPath. Пример: <filter name="xPathFilter" filterType="XPath" filterData="//*/location" />

Как можно заметить в табл. 14.1, используя тип `Custom`, вы можете реализовать собственный фильтр, просто наследуя абстрактный класс `MessageFilter`.

Служба маршрутизации реализуется как служба WCF, которую можно найти в сборке `System.ServiceModel.Routing`. Архитектура службы состоит из компонентов, приведенных ниже.

- Набор контрактов. `IDuplexSessionRouter`, `IRequestReplyRouter`, `ISimplexDatagramRouter` и `ISimplexSessionRouter`.
- Служба `RoutingService`, которая реализует доступные контракты маршрутизации.
- Режим службы, позволяющий определить таблицу фильтров, которую должна использовать служба маршрутизации.
- Параметры конфигурации маршрутизации, позволяющие определять фильтры, применяемые к вызовам клиентских конечных точек, и резервный список службы для обработки исключений.

Каждый определенный контракт позволяет обрабатывать сообщения, используя различные шаблоны обмена сообщениями. Вы должны выбрать правильный шаблон сообщения на основании реализации ваших внутренних служб. Фактически некоторые шаблоны сообщений могут работать неправильно. Что будет, если шаблон ответа на запрос, например Http, ждет ответного сообщения, которое никогда не будет получено при односторонней связи?

Как представлено в листинге 14.16, настройка службы маршрутизации действительно проста, поскольку она очень похожа на любую другую службу WCF. Используя файл конфигурации, создайте раздел службы.

Листинг 14.16. Настройка службы маршрутизации


```
<system.serviceModel>
    <services>
        <service behaviorConfiguration="routingBehavior"
                name="System.ServiceModel.Routing.RoutingService">
            <host>
                <baseAddresses>
                    <add baseAddress=" http://localhost:9001/routing%22/>
                </baseAddresses>
            </host>

            <endpoint address=""
                binding="basicHttpBinding"
                name="requestReplyEndpoint"
                contract="System.ServiceModel.Routing.IRequestReplyRouter" />
        </service>
    </services>
<system.serviceModel>
```

Режим `routingBehavior` просто определяет название таблицы маршрутизации, которую должна использовать служба (листинг 14.17).

Листинг 14.17. Настройка режима службы


```
<behaviors>
    <serviceBehaviors>
        <behavior name="routingBehavior">
            <routing filterTableName="routingTable" />
        </behavior>
    </serviceBehaviors>
</behaviors>
```

Теперь настройте список конечных точек, которые должна вызвать служба маршрутизации, действуя как клиент (листинг 14.18).

Листинг 14.18. Настройка клиентских конечных точек


```
<client>
    <endpoint name="HospitalServiceEndpoint"
            address="net.tcp://localhost:9050/hospitalservice"
            binding="netTcpBinding"
            contract="*" />
    <endpoint name="LabServiceEndpoint"
            address="net.tcp://localhost:9051/labservice"
            binding="netTcpBinding"
            contract="*" />
</client>
```

И наконец, чтобы обеспечить правильное выполнение службы маршрутизации, настройте правила маршрутизации. Как представлено в листинге 14.19, это можно сделать в разделе конфигурации `routing`.

Листинг 14.19. Настройка правил маршрутизации


```
<routing>

    <filters>
        <filter name="LabFilter"
            filterType="Action"
            filterData="http://healthcare/lab" />
        <filter name="HospitalFilter"
            filterType="Action"
            filterData="http://healthcare/hospital" />
    </filters>

    <filterTables>
        < filterTable name="routingTable">
            <add filterName="HospitalFilter"
                endpointName="HospitalServiceEndpoint"/>
            <add filterName="LabFilter"
                endpointName="LabServiceEndpoint" />
        </filterTable>
    </filterTables>
</routing>
```

В этом разделе вы создали и назвали два фильтра, причем каждый на основании действия сообщения. В разделе `filterTable` вы настраиваете конечную точку, которую следует использовать, когда один из фильтров применяется к входящему сообщению.

Как уже упоминалось, при наследовании абстрактного класса `MessageFilter` можно также использовать специальный фильтр, чтобы изменить стандартную фильтрацию сообщений (листинг 14.20).

Листинг 14.20. Создание специального фильтра сообщений


```
class VersionBasedMessageFilter : MessageFilter
{
    private string MessageVersion;

    public VersionBasedMessageFilter(object filterData)
    {
        this.MessageVersion = filterData as string;
    }

    public override bool Match(System.ServiceModel.Channels.Message message)
    {
        return this.InnerMatch(message);
    }

    public override bool Match(System.ServiceModel.Channels.MessageBuffer buffer)
    {
        bool response;
        Message message = buffer.CreateMessage();
        try
        {
            response = this.InnerMatch(message);
        }
        finally
        {
            message.Close();
        }

        return (response);
    }

    private bool InnerMatch(System.ServiceModel.Channels.Message message)
    {
        return (message.Version.Envelope.ToString == this.MessageVersion);
    }
}
```

Чтобы использовать специальный фильтр, установите файл конфигурации, как показано в листинге 14.21.

Листинг 14.21. Разрешение применения специального фильтра сообщений


```
<routing>
    <filters>
        <filter
            name="LabFilter"
```

```
            filterType="Custom"
            customType="HealthRoutingService.Description.VersionBasedMessageFilter,
            HealthRoutingService" filterData="http://healthcare/lab" />
    </filters>
</routing>
```

Мостовое соединение протокола и безопасности

Данный компонент позволяет службе получать сообщения от клиента по протоколу любого вида (например, HTTP) или кодировать сообщения (SOAP 1.1) и вызывать внутренние службы с помощью совершенно другого протокола (например, net.tcp с SOAP 1.2), как показано на рис. 14.16.

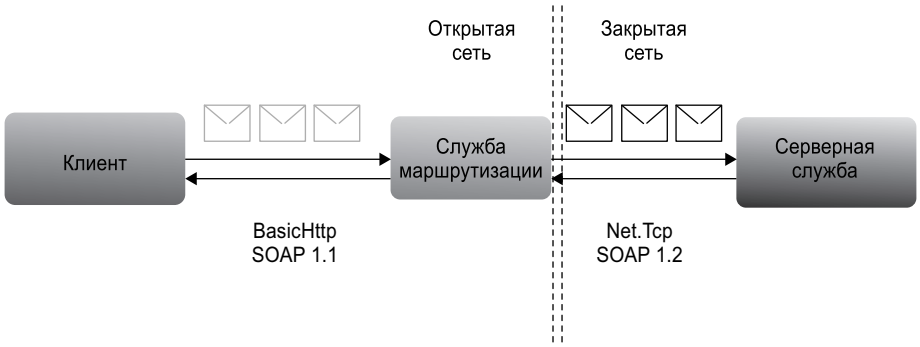


Рис. 14.16. Мостовое соединение

Для связи между клиентом и службой маршрутизации используйте одну привязку, а для связи между службой маршрутизации и серверными службами — другую. Это весьма удобно, если необходимо обеспечить или адаптировать связь с устаревшими клиентами, которые могли использовать только простую версию протокола HTTP, и новой версией службы, которая использует высокоэффективный протокол net.tcp.

Что касается протокола, то используйте своего рода механизм безопасности для службы маршрутизации, которая полностью отличается от серверной службы за счет отделения сертификатов, используемых клиентом, от внутренней системы идентификаторов.

Обработка ошибок

Еще одной возможностью службы маршрутизации является способность обслуживать исключения. Иногда, хотя ваше приложение и выполняется, служба может стать недоступной. В таком случае настройте службу маршрутизации так, чтобы задействовать список резервных служб.

Сначала настройте список клиентских конечных точек, используемых службой маршрутизации (листинг 14.22).

Листинг 14.22. Установка специальной конфигурации протокола net.tcp


```
<client>
    <endpoint name="LabServiceEndpoint"
        address="net.tcp://localhost:9052/labservice"
```

```
        binding="netTcpBinding"
        contract="*" />
    <endpoint name="LabServiceBackupEndpoint"
        address=" http://server1/labservice "
        binding="basicHttpBinding"
        contract="*" />
    <endpoint name="LabServiceBackupEndpoint"
        address="net.tcp://server2:9052/labservice"
        binding="netTcpBinding"
        contract="*" />
</client>
```

После настройки списка конечных точек и списка фильтров, которые вы хотите применить (пропущено в листинге 14.23), настройте раздел `filterTable`, чтобы определить конечную точку, которую должна использовать служба маршрутизации, когда применяется фильтр. В этом разделе вы сможете определить название резервного списка для поиска последовательности конечных точек, которые должна вызывать служба маршрутизации, если первая конечная точка передает исключение.

Листинг 14.23. Определение таблицы фильтра и имени резервного списка


```
<filterTables>
    < filterTable name="routingTable">
        <add filterName="LabFilter"
            endpointName="LabServiceEndpoint"
            backupList="labBackupList"/>
    </filterTable>
</filterTables>
```

И наконец, настройте список резервных конечных точек службы, которые должны относиться к существующей клиентской конечной точке, определенной в файле конфигурации (листинг 14.24).

Листинг 14.24. Определение конечных точек в резервном списке


```
    <backupLists>
        <backupList name="labBackupList">
            <add endpointName=" LabServiceBackupEndpoint "/>
            <add endpointName=" LabServiceBackup2Endpoint "/>
        </backupList>
    </backupLists>
</routing>
```

Хостинг на основе “облака”

В настоящей главе обсуждались службы хостинга в “классической” распространенной среде. В этом случае, как правило, вы либо являетесь владельцем серверов, на которых будет развернуто решение, либо вам, по крайней мере, известна непосредственно аппаратная инфраструктура. Мощность аппаратных средств выбирается на основании требований ваших решений. Поэтому, когда вам необходимо повысить мощь своей фермы серверов, приходится покупать новые аппаратные средства.

В среде "облаков", напротив, подробности об аппаратных средствах полностью абстрагированы от пользователя (или скорее от разработчика). Разработчик не обязан ничего знать об инфраструктуре. Концепция должна позволять увеличивать или уменьшать масштаб решения, задействуя по мере необходимости ресурсы по требованию в режиме платы за использованное. Если ваше решение при запуске требует меньше аппаратных ресурсов, то вы платите только за используемое хранилище, объем данных в транзакциях и компьютерное время, необходимое для поддержки доступности вашего приложения. Это существенно сокращает стоимость запуска, поскольку вы не должны вкладывать деньги в аппаратные средства и наем IT-специалистов.

Вся инфраструктура является полностью масштабируемой и имеет емкость, достаточную для хранения и обслуживания крупных объемов данных. Кроме того, среда "облака" обеспечивает надежность, позволяя сделать ваше решение доступным 24 часа в сутки 7 дней в неделю. Более того, система репликации узлов позволяет выполнять резервное копирование узла, если его неожиданная остановка нежелательна.

Безусловно, преимуществами среды "облаков" пользуются главным образом веб-приложения. Вы можете развернуть веб-приложение, которое предоставляет вашу службу, и получить все обсуждавшиеся ранее преимущества "облаков". Приложение, развернутое в инфраструктуре "облака", называется расположенным *в "облаке"* (in-the-cloud), в то время как "классическое" приложение называется *местным* (on-premises). Платформа Windows Azure — это решение корпорации Microsoft для приложений в "облаке". В следующих разделах рассматриваются возможности, предоставляемые инфраструктурой "облаков" службе WCF.

Хостинг служб WCF на платформе Windows Azure

Windows Azure — это набор технологий для среды "облаков". Как уже упоминалось, он позволяет развертывать веб-приложения, но обеспечивает также хранение в "облаке", чтобы обрабатывать такие данные, как таблицы, поля BLOB или очереди, а также базы данных SQL Azure Database, которые являются "облачной" версией сервера SQL Server. Другая возможность, предоставляемая платформой Azure, — это поддержка приложений и служб сетевых подключений с помощью службы Service Bus и службы управления доступом, которая выдает маркеры для интегрированной авторизации. Эти две возможности группируются в технологии AppFabric платформы Windows Azure. Она отличается от уже обсуждавшегося набора Windows Server AppFabric, и их не следует путать. Более подробная информация о технологии Azure платформы AppFabric приведена в следующем разделе.

Среда Windows Azure Computing позволяет развернуть приложение в "облаке". При развертывании своего приложения вы ничего не знаете об аппаратных средствах серверов (или узлов). Вы не знаете ни его расположения в веб-ферме, ни его IP-адреса и не можете обратиться к локальному ресурсу (например, к диску C: или чему-либо подобному). Вы просто разрабатываете свое решение и развертываете его в "облаке".

На платформе Azure есть два возможных типа приложений, называемых также ролями.

- Роль веб предназначена для интерфейсных решений, таких как веб-приложение.
- Роль рабочего процесса эквивалентна службе Windows. Разрабатывается для выполнения фоновых операций.

Эти две роли позволяют разграничить внешний интерфейс и связанные с ним серверные операции. В типичном случае роль веб заключается в создании очереди

сообщений, а роль рабочего процесса — в их обработке. Следует заметить, что вы не знаете, выполняется ли роль рабочего процесса на том же узле, что и роль веб. Таким образом, разграничение способствует балансу нагрузки и масштабированию вашего решения.

Если необходим хостинг службы WCF на платформе Windows Azure, использование роли веб является наилучшим решением. С точки зрения разработчика, нет никакого особенного различия между местным веб-приложением и веб-приложением в “облаке”. В среде разработки Visual Studio 2010 создайте службу WCF Azure Service с помощью шаблона Web Cloud Service, а затем добавьте свои службы и конечные точки как в любой другой веб-проект. Затем соберите свое решение и разверните его, используя созданные пакеты.

Служба Azure AppFabric Service Bus

В некоторых случаях доступ к вашим конечным точкам может быть затруднен в связи с динамическим присвоением IP-адресов, брандмауэрами или границами NAT. Для этих случаев служба Azure AppFabric Service Bus предоставляет специальный способ установки подключения. Любая служба, которая хочет получать сообщения, регистрирует свой адрес в службе Azure AppFabric Service Bus. Любое приложение, которое хочет посылать сообщения зарегистрированной службе, обращается к известной конечной точке, предоставленной службой Service Bus (рис. 14.17).

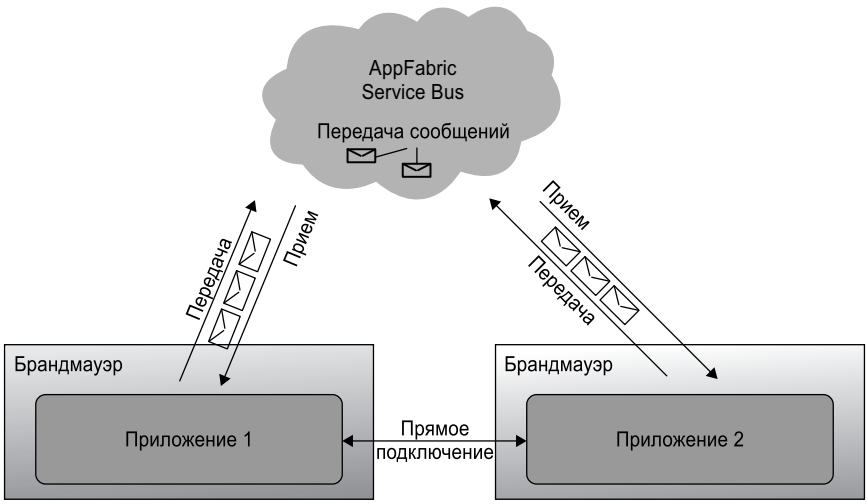


Рис. 14.17. Обмен сообщениями

Подключение между клиентом и службой начинается с использования режима передачи службы ретрансляции. Служба ретранслятора пытается сообщить клиенту информацию подключения к службе, а службе — информацию подключения к клиенту. В случае успеха соединение продолжается через прямое подключение. В противном случае для соединения по-прежнему используется режим передачи.

Передача через “облако”

Чтобы использовать службу AppFabric Service Bus с платформой WCF, вы не обязаны знать разные модели программирования. Можете определить контракты и создать

среду хоста точно таким же способом, как и для платформы WCF (см. главу 2). Единственное различие связано с адресом прослушивания (адресом назначения, если вы находитесь на стороне клиента) и используемой привязкой. На самом деле вам необходимо использовать специфическую привязку, которая использует одну из привязок, поддерживающих транспортный протокол ретранслятора. Новые привязки, доступные и устанавливаемые с комплектом .NET Services SDK, представлены в табл. 14.2.

Таблица 14.2. Привязки комплекта .NET Services

Привязка	Описание
`BasicHttpRelayBinding`	Версия привязки `BasicHttpBinding` для службы Service Bus
`NetEventRelayBinding`	Обеспечивает режим `Publish&Subscribe`. При использовании этой привязки сообщение может быть передано одной, либо нескольким прослушивающим службам, либо ни одной службе
`NetOnewayRelayBinding`	Соответствует одностороннему режиму подключения привязки `NetTcpBinding`
`NetTcpRelayBinding`	Основана на привязке `NetTcpBinding` и позволяет устанавливать подключения TCP через "облако"
`WebHttpRelayBinding`	Основана на привязке `WebHttpBinding` и позволяет устанавливать подключения в стиле RESTful
`WS2007HttpRelayBinding`	Основана на привязке `WS2007HttpBinding` и позволяет использовать для связи в "облаке" самые последние версии стандартных спецификаций WS-*
`WSHttpRelayBinding`	Основана на привязке `WSHttpBinding` и позволяет использовать для связи в "облаке" стандартные спецификации WS-*

Все эти привязки доступны в сборке `Microsoft.ServiceBus`. В комплекте November 2009 CTP of the Microsoft .NET Services SDK вы найдете эту сборку в папке `%programfiles%\Microsoft .NET Services SDK (Nov 2009 CTP)\Assemblies`.

Как уже упоминалось, чтобы обеспечить связь, вы должны создать специальный адрес. Как представлено в листинге 14.25, класс `ServiceBusEnvironment` позволит создать специальный адрес с помощью статического метода `CreateServiceUri`.

Листинг 14.25. Создание адреса для прослушивания службы Service Bus


```
Uri address = ServiceBusEnvironment.CreateServiceUri("sb", "carrental",
                "CarRentalRelayService");
```

Приведенный выше код создает следующий адрес URI: `sb://carrental.servicebus.windows.net/CarRentalRelayService/`. Первый параметр определяет схему URI, которая должна быть использована при создании адреса. Второй параметр определяет название решения AppFabric, третий — относительный адрес вашей службы.

Для создания адреса конечной точки используйте одну из привязок ретрансляции. И наконец, аутентифицируйте свое подключение, предоставив один из сертификатов, описанных в табл. 14.3.

Маркер SAML — это тип сертификата, используемого в интегрированном сценарии (см. главу 9); в данном случае он обеспечивает многократное использование существующей инфраструктуры. Сертификат `SharedSecret` — это маркер на основе общего ключа, такой как пароль или массив байтов, известных и клиенту, и службе; используется

для защиты связи. Сертификат SimpleWebToken, напротив, является действительно компактным зашифрованным маркером типа “ключ–значение”, защищен подписью и используется главным образом в архитектурах RESTful.

Таблица 14.3. Доступные сертификаты

Элемент	Описание
SAML	Для аутентификации службы AppFabric Service Bus используется маркер SAML (Security Assertion Markup Language — язык разметки утверждений безопасности)
SharedSecret	Для аутентификации службы AppFabric Service Bus используется общий ключ
SimpleWebToken	Для аутентификации службы AppFabric Service Bus используется простой маркер веб
Unauthenticated	Клиент не предоставляет никаких сертификатов

В следующем коде создайте новый класс ServiceHost, который отрабатывает переданную через “облако” версию службы CarRentalService.

Листинг 14.26. Хостинг службы при передаче через “облако”


```
ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Tcp;
Uri address = ServiceBusEnvironment.CreateServiceUri("sb", "carrental",
                "CarRentalRelayService");

NetTcpRelayBinding binding = new NetTcpRelayBinding();
binding.Security.Mode = EndToEndSecurityMode.None;

ServiceHost host = new ServiceHost(typeof(CarRentalService));
ServiceEndpoint endpoint =
    host.AddServiceEndpoint(typeof(ICarRentalService), binding, address);

TransportClientEndpointBehavior sharedSecretServiceBusCredential =
    new TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
    TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = "owner";
sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret =
    "4+X1iL29uQ4kJiDMBTQ9Sz9n8T16RTgL9SIpyGWKryc=";

endpoint.Behaviors.Add(sharedSecretServiceBusCredential);

host.Open();

Console.WriteLine("Waiting for incoming messages...");
Console.ReadLine();

host.Close();
```

Здесь использован сертификат типа SharedSecret, который нуждается в значениях IssuerName и IssuerSecret, чтобы аутентифицировать подключение к определенному приложению AppFabric Azure. Режим TransportClientEndpointBehavior используется для содержания сертификатов и должен быть установлен для конечной точки службы.

На клиентской стороне код подобен, как показано в листинге 14.27.

Листинг 14.27. Вызов на клиенте службы через "облако"

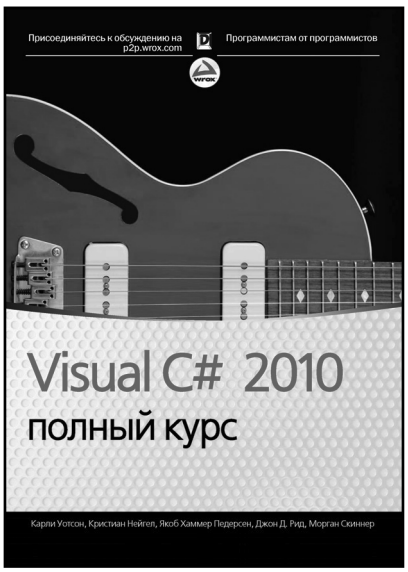

```
ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

Uri address = ServiceBusEnvironment.CreateServiceUri("sb", "carrental",
        "CarRentalRelayService");

NetTcpRelayBinding binding = new NetTcpRelayBinding();
binding.Security.Mode = EndToEndSecurityMode.None;

TransportClientEndpointBehavior sharedSecretServiceBusCredential =
        new TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
        TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = "owner";
sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret =
        "4+X1iL29uQ4kJiDMBTQ9Sz9n8T16RTgL9SIpyGWKryc=";

ServiceEndpoint endpoint = new ServiceEndpoint(
        ContractDescription.GetContract(typeof (ICarRentalService)),
        binding,
        new EndpointAddress(address));
endpoint.Behaviors.Add(sharedSecretServiceBusCredential);

ChannelFactory<ICarRentalService> channelFactory =
        new ChannelFactory<ICarRentalService>(endpoint);
ICarRentalService ps = channelFactory.CreateChannel();
```

Вы могли обратить внимание на применение параметра `ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http`. Это связано с тем, что вы можете устанавливать разные протоколы подключения для службы отправителя и службы получателя.

Предметный указатель

VISUAL C# 2010 ПОЛНЫЙ КУРС

Карли Уотсон и др.



www.dialektika.com

Этот исчерпывающий источник для начинающих поможет подготовиться к работе с новым выпуском языка программирования C#. Книга предлагает исчерпывающее описание синтаксиса C#. Читатели узнают о таких фундаментальных основах, как переменные, управление потоком выполнения и объектно-ориентированном программировании. Кроме того, будет показано, как строить Windows- и веб-приложения, формы Windows и работать с данными. Пошаговые упражнения позволят лучше усвоить материал, предлагаемый в каждой главе. Прочтя эту книгу, читатели сразу же смогут приступить к написанию собственного кода для решения разнообразных реальных задач.
Книга рассчитана на начинающих программистов, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1699-0 **в продаже**

VISUAL C++ 2010 ПОЛНЫЙ КУРС

Айвор Хортон

www.dialektika.com

По существу, в этой книге рассматриваются две обширные темы: язык программирования C++ и программирование приложений Windows с использованием MFC или .NET Framework. Прежде чем вы сможете разработать полнофункциональное приложение Windows, необходимо приобрести хороший уровень знаний языка C++, поэтому упражнения здесь на первом месте.

В первой части книги поэтапно изложены основные темы программирования на языке C++. Вы изучите синтаксис и использование базового языка C++, а также приобретете уверенность и опыт применения его на практике. Модификацию C++/CLI базового языка C++ вы также изучите на практических примерах.

Кроме того, вы узнаете о мощных инструментальных средствах, предоставляемых стандартной библиотекой шаблонов STL, для базовой версии языка C++ и версии C++/CLI. Одна из глав посвящена библиотеке шаблонов для параллельных вычислений, которая позволяет использовать мощь многоядерных PC для приложений с интенсивными вычислениями.

ISBN 978-5-8459-1698-3 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2010 И ПЛАТФОРМА .NET 4
5-е издание

Эндрю Троелсен

www.williamspublishing.com

Версия .NET 4 привнесла множество новых API-интерфейсов в библиотеках базовых классов, а также новых синтаксических конструкций в языке C#.
В этой книге вы найдете полное описание всех нововведений в характерной для автора дружественной к читателю манере. Помимо общих вопросов, подробно рассматривается среда Dynamic Language Runtime (DLR); библиотека Task Parallel Library (TPL, включая PLINQ); технология ADO.NET Entity Framework (а также LINQ to EF); расширенное описание API-интерфейса Windows Presentation Foundation (WPF); улучшенная поддержка взаимодействия с COM.
В книге рассматриваются следующие темы

- Особенности платформы .NET 4 и языка Visual C# 2010
- Детали технологии .NET – лидера в производстве современного программного обеспечения
- Полезные советы по разработке от эксперта в .NET, который изучает эту платформу, начиная с ее первой версии
- Полное описание технологий WPF, WCF и WF, поддерживаемых ядром платформы .NET

ISBN 978-5-8459-1682-2

в продаже