Дэниел Мол

Создание облачных, мобильных и веб-приложений на F#

Daniel Mohl

Building Web, Cloud, and Mobile Solutions with F#

O'REILLY®

Дэниел Мол

Создание облачных, мобильных и веб-приложений на F#

ДМК Пресс
издательство
Москва, 2013

УДК 004.432.42F#
ББК 32.973-018.1
М74

Мол Д.

М74 Создание облачных, мобильных и веб-приложений на F#. – М.: Пер. с англ. Киселева А. Н. ДМК Пресс, 2013. – 208 с.: ил.

ISBN 978-5-94074-924-0

Книга рассказывает о ключевых аспектах создания облачных, мобильных и веб-решений на языке F# в комбинации с различными технологиями для платформы .NET. На практических примерах демонстрируется, как решать проблемы конкуренции, асинхронного выполнения и другие, встречающиеся на стороне сервера. Вы узнаете, как повысить свою продуктивность с помощью языка F#, интегрируя его в существующие веб-приложения или используя его для создания новых проектов.

Опытные разработчики для .NET узнают, как этот выразительный язык функционального программирования помогает писать надежные и простые в сопровождении решения, легко масштабируемые и способные адаптироваться для работы на самых разных устройствах.

Издание предназначено для программистов разной квалификации, желающих использовать возможности функционального программирования в своих проектах.

УДК 004.432.42F#
ББК 32.973-018.1

ISBN 978-1-449-33376-8 (анг.)
ISBN 978-5-94074-924-0 (рус.)

Содержание

Предисловие 10
Для кого эта книга 11
Что необходимо для опробования примеров 11
Структура книги 12
Типографские соглашения 13
Использование программного кода примеров 14
Safari® Books Online 15
Как с нами связаться 15
Благодарности 16

Глава 1. Создание веб-приложений для ASP.NET MVC 4 на языке F# 17
Шаблоны проектов F# ASP.NET MVC 4 18
Поиск и установка шаблонов 19
Проект на C# 20
Проект на F# 21
Global.fs 21
HomeController.fs 23
Контроллеры и модели на F# 24
Контроллеры 25
Модели 26
Взаимодействие с базой данных 28
Entity Framework 28
Извлечение данных 31
Извлечение данных с использованием поставщиков типов 32
Использование преимуществ F# 34
Переход на функциональную парадигму 34
Конвейеры и частичное применение функций 36
Создание более функционального контроллера 38

Упрощение за счет сопоставления с образцом 40
Дополнительные темы и понятия 44
Улучшение времени отклика с помощью асинхронных операций 44
Кеширование с применением MailboxProcessor 46
Сообщения, как значения типа размеченного объединения 47
Основной агент 48
Использование агента CacheAgent 49
Шина сообщений 51
SimpleBus 52
Публикация сообщений 54
Извлечение сообщений 56
Стиль продолжений 57
Создание собственных вычислительных выражений 58
В заключение 60

Глава 2. Создание веб-служб на языке F# 62
Установка шаблона проекта WCF 63
Исследование получившегося решения 64
Использование службы 67
Погружение в записи 72
Создание службы ASP.NET Web API 73
Анализ шаблона 74
Взаимодействие с HTTP-службой 78
С использованием объекта HttpClient 79
Поставщик типов JSON 82
Прежде чем покинуть ASP.NET Web API 83
Другие веб-фреймворки 84
Service Stack 84
Nancy 87
Frank 90
Тестирование своих творений 94
Подготовка 94
Улучшение тестов с применением F# 97
FsUnit 99
Unquote 101
NaturalSpec 102
В заключение 104

Глава 3. К облакам! Использование преимуществ Azure 105

Создание и развертывание приложений F# на платформе Azure 106
Создание рабочей роли на F# 108
Введение в библиотеку Fog 109
Взаимодействие с хранилищами данных Azure 110
Большие двоичные объекты 110
Таблицы 112
Служба хранения очередей 114
SQL Azure 115
Использование преимуществ Azure Service Bus 116
Очереди 116
Темы 117
Аутентификация и авторизация 119
Аутентификация и авторизация с применением ACS 120
Аутентификация на основе заявок 121
Авторизация на основе заявок 122
Создание масштабируемых приложений 123
Создание веб-роли 124
PlaceOrderCommand 126
Рабочие роли 127
Рабочая роль SQL Azure 128
Последние штрихи 130
Кеширование 131
CDN и автоматическое масштабирование 132
Блистательные примеры на F# 133
{m}brace 134
Cloud Numerics 135
Hadoop MapReduce для .NET 136
В заключение 136

Глава 4. Создание масштабируемых мобильных и веб-приложений 137

Масштабирование с применением веб-сокетов 138
Пример использования веб-сокетов на платформе .NET 4.5 и IIS 8 139
Создание сервера веб-сокетов с помощью Fleck 144

SignalR 147
Пример создания постоянного соединения 148
Клиент на JavaScript 149
Клиент на F# 150
Пример создания хаба 150
Серверная сторона 151
Клиентская сторона 152
Обретаем мобильность 153
Способ на основе jQuery Mobile 153
Добавляем поддержку Windows Phone 155
Объединение F# и NoSQL 158
MongoDB 159
RavenDB 162
CouchDB 163
В заключение 165

Глава 5. Разработка интерфейсов в функциональном стиле 166

Подготовка почвы 167
Знакомство с LiveScript 168
Преимущества 168
Применение 169
Пример 171
Исследуем Pit 173
Преимущества 174
Применение 175
Пример 176
Погружение в WebSharper 179
Преимущества 180
Применение 181
Пример 182
В заключение 184

Приложение А. Полезные инструменты и библиотеки 186

FAKE (F# Make) 186
NuGEt 186
Основы использования 187
Полезные NuGet-пекты 188
ExpectThat 192

Приложение B. Полезные веб-сайты 194
fssnip.net 194
tryfsharp.org 194
Visual Studio Gallery 195
jQueryMobile.com 195

Приложение C. Клиентские технологии, совместимые с F# 196
CoffeeScript 196
Sass 197
Underscore.js 200
Об авторе 201

Предметный указатель 202

Предисловие

Если проанализировать самые последние веянья в развитии информационных технологий, можно увидеть, что основным направлением является создание облачных, мобильных и веб-решений, масштабируемых в широких пределах и связанных с управлением большими объемами данных. С появлением этих направлений возникла потребность в инструментах, позволяющих специалистам, таким как вы или я, создавать собственные решения в этой области. Что для этого нужно? Какие архитектуры, инструменты, языки и технологии можно использовать для разработки программ, способных выполняться на самых разных устройствах и легко масштабироваться, и при этом обеспечить высокую надежность решений, простоту их сопровождения, тестирования и многократного использования?

Существует множество инструментов, отвечающих нашим потребностям, но для решения наших задач в полном объеме, их возможностей оказывается недостаточно. Чтобы получить максимальную отдачу, необходим язык, специально предназначенный для преодоления сложностей, возникающих в описанных областях разработки. Он должен иметь встроенные средства для преодоления проблем, связанных с конкуренцией, выполнением асинхронных операций и большими объемами данных, и при этом прозрачно интегрироваться с другими языками, технологиями и инструментами, лучше подходящих для решения других задач. К счастью, такой язык существует и называется F#.

В этой книге я покажу, как использовать язык F# для реализации ключевых элементов облачных, мобильных и веб-приложений, и решения упомянутых проблем. Выразительность, широта возможностей, лаконичность и функциональная природа языка F#, в сочетании с уже известными вам технологиями, такими как ASP.NET MVC, ASP.NET Web API, WCF, Windows Azure, HTML5, CSS3, JavaScript, jQuery и jQuery Mobile, позволят вам создавать удивительные приложения, не только соответствующие, но и превосходящие текущие и будущие требования к ним.

Для кого эта книга

Эта книга предназначена для специалистов с опытом работы в .NET, слышавших о преимуществах F#, имеющих хотя бы общее представление о его синтаксисе, и желающих узнать, как объединять F# с другими технологиями для создания облачных, мобильных и веб-приложений. Если вы совершенно не знакомы с F#, я предлагаю заглянуть в другие книги, описывающие основы программирования на этом языке, такие как книга Криса Смита (Chris Smith) «Programming F#, 3.0» (O'Reilly)[1]. Если вы не знакомы с другими платформами и фреймворками, упоминаемыми в этой книге, такими как ASP.NET MVC, WCF, ASP.NET Web API, Windows Azure, HTML, CSS и/или jQuery Mobile, их описание можно найти во множестве других книг, где вы сможете почерпнуть всю необходимую информацию.

Что необходимо для опробования примеров

Большая часть примеров для этой книги была создана с помощью Visual Studio 2012. Для опробования примеров я рекомендую использовать версию Visual Studio 2012 Professional или выше; однако, большинство примеров будет также работать в среде F# Tools для Visual Studio Express 2012 for Web, анонсированной 12 сентября 2012 в блоге команды разработчиков F#[2]. Загрузить F# Tools для Visual Studio Express 2012 for Web можно по адресу: http://www.microsoft.com/web/gallery/install.aspx?appid=FSharpVWD11. В зависимости от целевой платформы или фреймворка, может потребоваться установить следующие инструменты:

- ASP.NET MVC 4, можно загрузить по адресу: http://www.asp.net/mvc/mvc4.
- Windows Azure SDK и Developer Tools, можно загрузить по адресу: http://www.windowsazure.com/en-us/develop/net/.

Дополнительные библиотеки и инструменты, которые могут потребоваться, упоминаются в соответствующих главах.

---

[1] Крис Смит, «Программирование на F#», ISBN: 978-5-93286-199-8, Символ-Плюс, 2011. – *Прим. перев.*

[2] http://bit.ly/fsharp-blog.

Структура книги

В этой книге рассказывается обо всем, что необходимо знать, чтобы приступить к разработке облачных, мобильных и веб-приложений на языке F#. Кроме того, здесь описывается множество новейших технологий, платформ и библиотек, таких как Windows Azure, jQuery Mobile, SignalR, CouchDB, RavenDB, MongoDB и других. Ниже подробнее описывается, что вы увидите в каждой главе.

Глава 1, «Создание веб-приложений для ASP.NET MVC 4 на языке F#»

В этой главе рассказывается обо всем, что необходимо знать, чтобы приступить к созданию веб-приложений на языке F# с использованием фреймворка ASP.NET MVC 4 на стороне сервера. Здесь также демонстрируются некоторые дополнительные возможности и особенности языка F#, позволяющие писать более элегантный код.

Глава 2, «Создание веб-служб на языке F#»

Эта глава знакомит с инструментами и понятиями, используемыми при создании веб-служб различных типов, включая службы WCF SOAP и HTTP, и особенностями взаимодействий с некоторыми маленькими веб-фреймворками. Здесь также рассказывается об инструментах и приемах модульного тестирования этих веб-служб.

Глава 3, «К облакам! Использование преимуществ Azure»

Эта глава проведет вас через создание веб-приложений и веб-служб на языке F#, выполняющихся под управлением Windows Azure. Дополнительно в ней даются примеры на F# взаимодействий с различными библиотеками Azure. В конце главы будут представлены некоторые замечательные библиотеки и фреймворки на F# для использования на платформе Azure.

Глава 4, «Создание масштабируемых мобильных и веб-приложений»

В этой главе более подробно освещаются вопросы совместного использования F# с другими технологиями для создания масштабируемых решений, позволяющими повторно использовать мобильные и веб-интерфейсы. Глава включает информацию и примеры использования веб-сокетов, библио-

теки SignalR, различных баз данных NoSQL, и многих других механизмов.

Глава 5, «Разработка интерфейсов в функциональном стиле»

Эта глава знакомит с LiveScript, Pit и WebSharper – инструментами, позволяющими, кроме всего прочего, писать клиентский код в функциональном стиле. Они делают возможным создавать комплексные веб-стеки с использованием концепций функционального программирования. Для каждого инструмента перечисляются его преимущества, начальная информация и примеры использования.

В конце книги вы найдете несколько приложений с информацией, которая может вам пригодиться в освоении приемов разработки ультрасовременных облачных, мобильных и веб-приложений, но не относящаяся к темам, обсуждаемым в основных главах.

Приложение A, «Полезные инструменты и библиотеки»

Здесь перечисляются и коротко описываются некоторые инструменты, которые могут облегчить вам жизнь, как разработчика облачных, мобильных и веб-решений.

Приложение B, «Полезные веб-сайты»

Здесь приводятся ссылки на веб-сайты, предлагающие информацию о языке F#, а также инструменты и библиотеки, упоминаемые в книге.

Приложение C, «Клиентские технологии, совместимые с F#»

Здесь дается краткий обзор некоторых технологий, дополняющих F# при разработке мобильных и веб-приложений.

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Курсив применяется для выделения новых терминов, имен файлов и их расширений.

`Моноширинный шрифт`

Применяется для представления листингов программного кода, а также в основном тексте для выделения элементов про-

грамм, таких как имена переменных и функций, базы данных, типы данных, переменные окружения, инструкции и ключевые слова.

`Моноширинный жирный`

Используется для выделения команд или другого текста, которые должны вводиться пользователем.

Моноширинный наклонный

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Примечание. Так обозначаются советы, предложения и примечания общего характера.

Внимание. Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию, вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Building Web, Cloud, and Mobile Solutions with F# by Daniel Mohl (O'Reilly). Copyright 2013 Daniel Mohl, 978-1-449-33376-8».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Safari® Books Online

Safari Books Online (www.safaribooksonline.com) – это виртуальная библиотека, содержащая авторитетную информацию в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса.

Профессионалы в области технологии, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов для организаций, правительственных учреждений и физических лиц. Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesly Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и десятков других. За подробной информацией о Safari Books Online обращайтесь по адресу: http://www.safaribooksonline.com/.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги: http://oreil.ly/building-web.

Свои пожелания и вопросы технического характера отправляйте по адресу: bookquestions@oreilly.com.

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте: http://www.oreilly.com.

Ищите нас на Facebook: http://facebook.com/oreilly.

Следуйте за нами на Твиттере: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Прежде всего я хотел бы поблагодарить мою супругу Мелиссу (Melissa) и дочь Еву (Eva) за то, что поддерживали меня долгие часы сидения за компьютером, когда я работал над этой книгой.

Я также хочу сказать спасибо всем, кто перечислен ниже, за их труд, поддержку и помощь, и за все их лучшие качества!

- Дон Сайм (Don Syme) и все остальные члены команды разработчиков F#;
- Рейчел Румелиотис (Rachel Roumeliotis);
- Илайджа Манор (Elijah Manor);
- Стивен Свенсен (Stephen Swensen);
- Фахад Сухаиб (Fahad Suhaib);
- Райан Райли (Ryan Riley);
- Стеффен Форкман (Steffen Forkmann);
- Антон Таяновский (Anton Tayanovskyy);
- Адам Гранич (Adam Granicz).

Глава 1. Создание веб-приложений для ASP.NET MVC 4 на языке F#

Любая достаточно развитая технология неотличима от магии.

– Сэр Артур Чарльз Кларк (Arthur Charles Clarke)

Я всегда испытывал благоговение перед волшебством и с раннего детства обожал наблюдать за фокусниками. Повзрослев, я стал читать все книги подряд, какие только мог найти, описывающие секреты фокусов, изумлявших меня в течение стольких лет. Вскоре я поймал себя на мысли, что изучать секреты фокусов мне нравится больше, чем смотреть их.

Как заметил сэр Артур Чарльз Кларк, технологии часто сравнимы с магией. Возможно поэтому я так полюбил технические науки. Язык F# относится к этой категории даже больше, чем другие языки, которые мне приходилось использовать в моей карьере программиста. Особенности этого языка открывают такие широкие возможности, что их с полным основанием можно назвать волшебством. Иногда бывает трудно определить, как лучше применить это волшебство на практике для создания еще более масштабируемых облачных, мобильных и веб-приложений, работающих еще лучше, еще быстрее. Эта книга покажет вам, как использовать все возможности языка F# для решения повседневных задач разработки.

В этой главе мы начнем свое путешествие с исследования возможности интеграции F# с фреймворком ASP.NET MVC 4. Здесь вы узнаете, как создать проект, как вести разработку на F# с использованием фреймворка ASP.NET MVC и как применять некоторые дополнительные возможности языка F# для улучшения программного кода. Мы также рассмотрим некоторые темы и приемы, не имею-

щие прямого отношения к ASP.NET MVC 4, но часто используемые вместе с этим фреймворком. На протяжении всей главы мы будем снимать покров тайны с особенностей F#, которые на первый взгляд могут показаться магическими.

В последующих главах мы будем знакомиться с другими платформами, технологиями, библиотеками и механизмами, которые можно использовать в программах на языке F# для создания ультрасовременных облачных, мобильных и веб-решений.

Шаблоны проектов F# ASP.NET MVC 4

О выходе предварительной версии ASP.NET MVC 4 для разработчиков было объявлено после конференции Build Conference во второй половине 2011 года. В феврале 2012 было объявлено о выходе бета-версии ASP.NET MVC 4 и в конце мая 2012 последовала предвыпускная (release candidate) версия. Версия 4 принесла множество улучшений и усовершенствований в и без того полнофункциональный фреймворк ASP.NET MVC. Дополнительную информацию о ASP.NET MVC 4 можно найти на веб-сайте проекта http://www.asp.net/mvc/mvc4.

Самый эффективный способ интеграции F# с фреймворком ASP.NET MVC 4 – использовать преимущества разделения задач, присущие шаблону проектирования «модель–представление–контроллер» (Model–View–Controller, MVC). Это разграничение можно с успехом использовать для усиления экосистемы C# возможностями языка F#. В случае с фреймворком ASP.NET MVC, это достигается путем создания проекта C# ASP.NET MVC, где будут сосредоточены представления и все программные компоненты, выполняющиеся на стороне клиента, и проекта на F#, для реализации моделей, контроллеров и других компонентов, выполняющихся на стороне сервера. На рис. 1.1 показана реализация типичного шаблона проектирования MVC в ASP.NET MVC с обозначением типов компонентов.

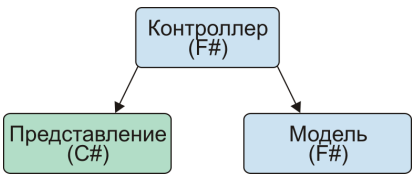

Рис. 1.1. Шаблон проектирования MVC с обозначением типов компонентов

Конечно, приложение с подобной структурой можно создать и вручную, но такой подход быстро становится утомительным. Кроме того, рутинные подготовительные операции являются дополнительным барьером на пути к использованию F# в приложениях на основе ASP.NET MVC. Чтобы помочь устранить эти проблемы, был создан шаблон проекта, доступный в галерее шаблонов проектов Visual Studio Gallery.

Примечание. Для тех, кто по каким-то причинам не может использовать шаблоны проектов ASP.NET MVC 4, в галерее также присутствуют шаблоны ASP.NET MVC 3 и ASP.NET MVC 2. Список большинства доступных шаблонов можно найти по адресу: http://bit.ly/allfsprojecttemplates.

Поиск и установка шаблонов

Благодаря галерее шаблонов проектов Visual Studio Gallery, поиск и установка шаблонов проектов F# ASP.NET MVC 4 выполняются проще некуда. Просто запустите мастер создания нового проекта любым способом, по вашему выбору, – я предпочитаю комбинацию клавиш **Ctrl+Shift+N** – Выберите пункт **Online (В Интернете)** в левой панели, введите текст «fsharp mvc4» в строке поиска в правом верхнем углу окна, выберите шаблон «F# C# MVC 4» и щелкните на кнопке **OK**. На рис. 1.2 изображено окно мастера создания нового проекта в момент, непосредственно перед щелчком на кнопке **OK**.

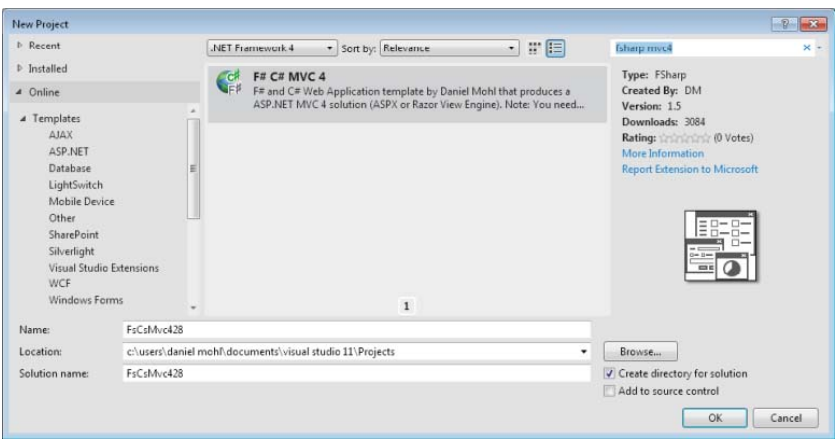

Рис. 1.2. Поиск шаблона проекта в галерее Visual Studio Gallery

Примечание. Описанный порядок действий можно использовать каждый раз при создании нового проекта F# ASP.NET MVC 4, но в действительности достаточно выполнить их один раз. После первичной установки новый шаблон будет доступен в категории «Installed» («Установленные»), в панели слева. Шаблону будет дано имя «F# and C# Web Application (ASP.NET MVC 4)» и вы сможете выбирать его в категории Visual F#→ASP.NET.

После щелчка на кнопке **OK** появится диалог (изображенный на рис. 1.3), где можно выбрать тип приложения и механизм представлений (раскрывающийся список **View Engine**), а также необходимость включения дополнительного проекта, где будут находиться модульные тесты. После выбора нужных параметров щелкните на кнопке **OK**. В результате будут созданы все необходимые проекты и установлены пакеты NuGet. В большинстве оставшихся примеров в этой главе будет предполагаться, что в процессе создания приложения был выбран механизм представлений Razor.

Рис. 1.3. Диалог мастера создания проекта F# ASP.NET MVC

Проект на C#

Если прежде вам приходилось создавать проекты ASP.NET MVC только на C#, приложение C#, созданное выше, покажется вам очень знакомым. В действительности рассматриваемый проект имеет всего три основных отличия:

1. Отсутствует папка *Controllers*.
2. Отсутствует папка *Models*.
3. Файл *Global.asax* не имеет соответствующего ему файла *Global.asax.cs*.

Главная причина этих отличий в том, что перечисленные элементы были перемещены в проект на F#, сгенерированный вместе с данным проектом на C#, но подробнее проект на F# будет рассматриваться в следующем разделе. Файл *Global.asax* не представляет большого интереса. В нем определен лишь один метод для связи с классом на F#. В следующем фрагменте показано содержимое файла *Global.asax*:

```
<%@ Application Inherits="FsWeb.Global" Language="C#" %>
<script Language="C#" RunAt="server">

  // Определение метода Application_Start, вызывающего метод Start
  // класса System.Web.HttpApplication, который наследуется классом Global.
  protected void Application_Start(Object sender, EventArgs e) {
       base.Start();
  }

</script>
```

Проект на F#

Если в диалоге мастера создания проекта (рис. 1.3) был выбран шаблон «Empty Project» (пустой проект), получившийся проект на F# будет очень прост. В проект автоматически будут добавлены все необходимые ссылки на сборки MVC и два файла *.fs*: *Global.fs* и *HomeController.fs*. Я уже коротко упоминал файл *Global.fs* и уверен, что вы уже догадались, что содержит файл *HomeController.fs*. Рассмотрим их подробнее в этом разделе.

Global.fs

Как уже упоминалось, файл *Global.fs* содержит большую часть кода, который обычно находится в файле *Global.asax.cs*, но с некоторыми особенностями, характерными для F#. Первое, что можно в нем заметить, – определение типа `Route`. Это *тип записи* на языке F#, предназначенный для создания определений маршрутов. Типы записей по умолчанию являются неизменяемыми. Поэтому они хорошо согласуются с конкурентной природой Веб, не предполагаю-

щей хранения информации о состоянии. Подробнее о типах записей я буду рассказывать далее в этой книге. Тип `Route` объявлен, как показано ниже:

```
type Route = { controller : string
               action : string
               id : UrlParameter }
```

> **Примечание.** Тип `Route` используется только для определения стандартных маршрутов контроллер/действие/ID. Для определения маршрутов других видов необходимо создавать собственные типы.

За объявлением типа `Route` следует определение класса `Global`, который наследует класс `System.Web.HttpApplication`. Код в классе `Global` выглядит почти так же, как в определении аналогичного ему класса на языке C#, за исключением вызова метода `MapRoutes` и использования значимых пробелов вместо фигурных скобок для определения области видимости. Главное отличие в вызове метода `MapRoutes` напрямую связано с типом `Route`. Для передачи информации о маршрутах методу `MapRoutes`, благодаря механизму определения типов в языке F#, вместо нового анонимного типа создается новая запись типа `Route`. Такой синтаксис создания записей называется *выражение записи* (record expression). Ниже приводится определение класса `Global`, где выделен фрагмент, выполняющий создание записи типа `Route`:

```
type Global() =
     inherit System.Web.HttpApplication()

     static member RegisterRoutes(routes:RouteCollection) =
          routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
          routes.MapRoute("Default",
                          "{controller}/{action}/{id}",
                          { controller = "Home"; action = "Index"
                            id = UrlParameter.Optional } )

     member this.Start() =
          AreaRegistration.RegisterAllAreas()
          Global.RegisterRoutes(RouteTable.Routes)
```

HomeController.fs

Файл `HomeController.fs` содержит определение класса `HomeController`. Он наследует класс `Controller` и реализует единственное действие с именем `Index`. Подробнее о контроллерах будет рассказываться ниже в этой главе. Файл `HomeController.fs` содержит следующий код:

```
namespace FsWeb.Controllers

open System.Web
open System.Web.Mvc

[<HandleError>]
type HomeController() =
     inherit Controller()
     member this.Index () =
          this.View() :> ActionResult
```

Кого-то может смутить комбинация символов `:>`, выделенная в предыдущем примере. Эта последовательность обозначает приведение типа вверх (upcast) результата вызова `this.View()` к типу `ActionResult`. В данном примере приведение к типу `ActionResult` не является необходимостью, но может потребоваться в некоторых других случаях, поэтому разработчики добавили приведение типа вверх в шаблон с целью демонстрации. Если бы тип возвращаемого значения метода `Index` был определен явно:

```
member this.Index () : ActionResult = ...
```

тогда приведение типа следовало бы записать так:

```
upcast this.View()
```

Так как в данном конкретном случае приведение типа не требуется, этот метод можно упростить, как показано ниже:

```
member this.Index () =
     this.View()
```

Примечание. Проверка возможности приведения типа вверх (upcast) производится на этапе компиляции, чтобы гарантировать его допусти-

мость. Но возможность приведения типа вниз (downcast) (например, оператором `:?>`) может быть проверена только на этапе выполнения. Если есть вероятность, что приведение типа вниз может потерпеть неудачу, рекомендуется предварительно выполнять проверку типа с помощью выражения сопоставления (match expression). Выражение приведения типа вниз можно также заключить в инструкцию `try/with` и предусмотреть обработку исключения `Invalid CastException`, но такое решение менее эффективно, чем проверка типа.

Контроллеры и модели на F#

Основная цель этой книги состоит в том, чтобы показать, как лучше использовать F# в обширном стеке технологий, поэтому о контроллерах и моделях будет рассказываться намного больше, чем о представлениях. Язык F# обладает рядом уникальных особенностей, прекрасно подходящих для реализации различных аспектов контроллеров и моделей. С некоторыми из них я познакомлю вас в этом разделе, а в следующих расскажу о более совершенных возможностях.

Чтобы вам было проще, обсуждение контроллеров и моделей будет вестись на примере создания новой страницы в веб-приложении, при этом особое внимание будет уделяться коду, реализующему создание модели и контроллера. Эта страница будет отображать список простое представление списка jQuery Mobile, управляемое и заполняемое новым контроллером и моделью.

Сначала создадим новое представление. Для этого создайте в папке *Views* новую папку *Guitars* и добавьте туда новое представление ASP.NET MVC с именем *Index*. Не забудьте снять флажок **Use a layout or master page:** (Использовать макет или главную страницу) в диалоге мастера создания элемента представления ASP.NET MVC. Теперь можно изменить разметку представления, как показано ниже:

```
@model IEnumerable<FsWeb.Models.Guitar>
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet"
        href= "http://code.jquery.com/mobile/1.0.1/jquery.mobile-1.0.1.min.css" />
</head>

<body>
```

```
    <div data-role="page" data-theme="a" id="guitarsPage">
        <div data-role="header">
            <h1>Guitars</h1>
        </div>
        <div data-role="content">
        <ul data-role="listview" data-filter="true" data-inset="true">
            @foreach(var x in Model) {
                <li><a href="#">@x.Name</a></li>
               }
        </ul>
        </div>
    </div>
    <script src="http://code.jquery.com/jquery-1.6.4.min.js">
    </script>
    <script src= "http://code.jquery.com/mobile/1.0.1/jquery.mobile-1.0.1.min.js">
    </script>

    <script>
        $(document).delegate("#guitarsPage", 'pageshow', function (event) {
            $("div:jqmData(role='content') > ul").listview('refresh');
        });
    </script>
</body>
</html>
```

Примечание. Поскольку представление не является основной целью этого раздела, для простоты я поместил его целиком в один файл с расширением .cshtml. Обычно же код на JavaScript принято помещать в отдельный модуль (и, иногда, использовать дополнительные инструменты, такие как библиотека RequireJS, упрощающие загрузку и управление модулями JavaScript). Кроме того, может потребоваться создать отдельную страницу `Layout` для использования во всех страницах мобильного интерфейса. Дополнительно, фреймворк ASP.NET MVC 4 поддерживает ряд соглашений, в соответствии с которыми добавляет к именам представлений «`.Mobile`» или другое окончание, в зависимости от вида устройства. Поближе познакомиться с рекомендуемыми приемами создания представлений можно по адресу: http://www.asp.net/mvc/tutorials.

Контроллеры

Чтобы создать простейший контроллер для нового представления, добавьте в проект на F# новый файл с исходным кодом, дайте ему имя *GuitarsController.fs* и сохраните в нем следующий код:

```
namespace FsWeb.Controllers

open System.Web.Mvc
open FsWeb.Models

[<HandleError>]
type GuitarsController() =
     inherit Controller()
     member this.Index () =
          // Последовательность жестко определена, исключительно в
          // демонстрационных целях.
          // Будет удалена в будущем примере.
          seq { yield Guitar(Name = "Gibson Les Paul")
                yield Guitar(Name = "Martin D-28") }
          |> this.View
```

Выглядит очень похоже на `HomeController`, за исключением выражения последовательности (sequence expression) и *прямого конвейерного оператора* (pipe-forward). В этом примере выражение последовательности определяет коллекцию экземпляров модели `Guitar` для передачи представлению. В будущем примере эти «жестко зашитые» данные мы заменим обращением к хранилищу данных.

Второй интересный момент – использование прямого конвейерного оператора. В этом примере прямой конвейерный оператор используется для передачи последовательности экземпляров `Guitar` в аргументе модели перегруженному методу `View`, принимающему единственный аргумент `obj`.

> **Примечание.** Ключевое слово `obj` – это псевдоним типа `object` в языке F#. Подробнее о псевдонимах типов будет рассказываться в главе 2.

Модели

Модели могут быть экземплярами записей или классов. Стандартные записи языка F# отлично подходят для представления данных, доступных только для чтения, и обеспечивают простоту моделей. В версии F# 3.0 появился новый атрибут `CLIMutable`, превращающий записи языка F# в отличный выбор, когда данные также должны быть доступны для чтения/записи. Подробнее об атрибуте `CLIMutable` будет рассказываться в главе 4. Ниже приводится пример модели `Guitar`, сконструированной на основе записи:

```
namespace FsWeb.Models

type Guitar = { Id : Guid; Name : string }
```

> **Примечание.** В версиях F#, ниже F# 3.0, записи также можно было использовать для представления изменяемых данных, хотя и с некоторыми сложностями, обусловленными отсутствием у записей конструкторов без параметров. Более удачным решением этой проблемы (до версии F# 3.0) было использование собственного механизма связывания моделей (model binder).

Второй способ определения моделей на языке F# основан на классах. Пример контроллера в предыдущем разделе предполагает, что используется подход на основе класса. Следующий пример демонстрирует, как определить класс модели `Guitar` (этот класс, как и большая часть примеров в этой книге, был написан с учетом особенностей версии F# 3.0; в версии F# 2.0 синтаксис может несколько отличаться, потому что автоматические свойства (auto-properties) появились только в версии F# 3.0):

```
namespace FsWeb.Models

type Guitar() =
    member val Name = "" with get, set
```

В класс модели допускается добавлять любые атрибуты аннотаций данных (Data Annotations). В следующем примере я добавил атрибут `Required` к свойству `Name`:

```
open System.ComponentModel.DataAnnotations

type Guitar() =
    [<Required>] member val Name = "" with get, set
```

> **Примечание.** В данном примере атрибуту `Required` не передается значение, но такая возможность может пригодиться во многих случаях, где поддерживается возможность изменения из пользовательского интерфейса.

Ниже приводится модель, которая будет использоваться в примере на основе фреймворка Entity Framework, в следующем разделе:

```
namespace FsWeb.Models

open System
open System.ComponentModel.DataAnnotations

type Guitar() =
    [<Key>] member val Id = Guid.NewGuid() with get, set
    [<Required>] member val Name = "" with get, set
```

Взаимодействие с базой данных

Если запустить веб-приложение прямо сейчас, вы увидите простую страницу, отображающую список названий гитар. Но в этом мало проку, потому что данные жестко определены в исходном коде. К счастью, в F# имеется несколько средств на выбор, позволяющих обращаться к базам данных для сохранения и извлечения данных.

Entity Framework

Фреймворк Entity Framework (EF) – это, пожалуй, один из наиболее распространенных в ASP.NET MVC инструментов взаимодействий с базами данных SQL Server и его распространение продолжается, особенно теперь, когда EF поддерживает поход «сначала код» (code-first). Шаблон F#/C# ASP.NET MVC 4 уже добавил ссылки на сборки, необходимые для работы с EF, поэтому можно сразу приступать к использованию фреймворка и создать класс, наследующий класс `DbContext`, как показано в следующем примере:

```
namespace FsWeb.Repositories

open System.Data.Entity
open FsWeb.Models

type FsMvcAppEntities() =
    inherit DbContext("FsMvcAppExample")

    do Database.SetInitializer(new CreateDatabaseIfNotExists<FsMvcAppEntities>())

    [<DefaultValue()>] val mutable guitars : IDbSet<Guitar>
    member x.Guitars with get() = x.guitars and set v = x.guitars <- v
```

Здесь не происходит ничего особенного. Мы просто использовали некоторые стандартные особенности EF API для определения множества `IDbSet` гитар, и создали свойство `Guitars` с методами чтения и записи.

> **Примечание.** Поближе познакомиться с EF API можно по адресу: http://bit.ly/efcodefirstwalkthrough.

Теперь необходимо добавить класс репозитория, чтобы получить возможность извлекать информацию о гитарах из базы данных.

> **Примечание.** Технически класс репозитория не нужен, но многие считают его удобным, а его применение в приложениях стало стандартной практикой.

Следующий пример содержит определение класса `GuitarsRepository`:

```
namespace FsWeb.Repositories

type GuitarsRepository() =
     member x.GetAll () =
     use context = new FsMvcAppEntities()
     query { for g in context.Guitars do
             select g }
     |> Seq.toList
```

> **Примечание.** Если вы вынуждены использовать EF в F# 2.0, синтаксис запроса в примере выше не будет работать (поддержка запросов появилась только в версии F# 3.0). Аналогичную возможность в версии F# 2.0 можно получить, если установить пакет NuGet с именем `FSPowerPack.Linq.Community`, открыть `Microsoft.FSharp.Linq.Query` *и заменить код запроса следующим:*
>
> ```
> query <@ seq { for g in context.Guitars -> g } @> |> Seq.toList
> ```
>
> Эта реализация из F# PowerPack использует особенность языка F# с названием «цитируемые выражения» (quoted expressions), позволяющую сгенерировать абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) и обработать его. Цитируемые выражения применяются для самых разных нужд, но чаще всего они используются, чтобы сгенерировать код на F# или других языках.

Первое, что делает метод `GetAll`, – создает экземпляр `DbContext`. Обратите внимание, что вместо стандартного ключевого слова `let` здесь

используется `use`. Это гарантирует своевременное удаление объекта по окончании его использования – напоминает обертывание кода инструкцией `using` в языке C#, но не требует обертывать какой-либо дополнительный код.

Затем метод `GetAll` выполняет запрос к базе данных. Для этой цели используется синтаксис *запросов*, появившийся в версии F# 3.0 и упрощающий операции с данными. Несмотря на то, что синтаксис запросов выглядит как новая особенность компилятора, в действительности этот механизм реализован на основе особенности языка F# с названием *вычислительные выражения* (computation expressions). Далее в этой главе я покажу, как создавать собственные вычислительные выражения. А в следующем разделе мы детально исследуем вычислительные выражения запросов.

Теперь осталось лишь заменить жестко определенные данные в первоначальной версии действия `Index` в классе `GuitarsController`:

```
[<HandleError>]
type GuitarsController(repository : GuitarsRepository) =
    inherit Controller()
    new() = new GuitarsController(GuitarsRepository())
    member this.Index () =
        repository.GetAll()
        |> this.View
```

Это изменение не только упростило код (особенно действие `Index`), но и добавило сложностей в виде определения нового перегруженного конструктора. Он служит следующим целям.

- Позволяет передавать репозиторий в конструктор, открывая тем самым путь к использованию принципа *инверсии управления* (Inversion of Control, IoC) в контейнерах. Ради простоты в предыдущий пример были включены не все изменения, необходимые для оптимального использования принципа IoC в контейнерах.
- Делает контроллер более удобным для тестирования. С добавлением перегруженного конструктора появляется возможность передавать репозиторий фиктивного класса, чтобы протестировать действия контроллера без подключения к фактической базе данных.

Поскольку по умолчанию фреймворк ASP.NET MVC требует наличие у контроллера конструктора без параметров, необходимо также добавить следующую строку кода:

```
new() = new GuitarsController(GuitarsRepository())
```

Она объявляет необходимый конструктор, который вызывает основной конструктор с новым объектом `GuitarsRepository`.

В заключение, прежде чем испытать возможность взаимодействий с базой данных, добавьте в файл *web.config*, находящийся в проекте C# Web Application, соответствующую строку подключения, такую как:

```
<add name="FsMvcAppExample"
     connectionString="YOUR CONNECTION STRING"
     providerName="System.Data.SqlClient" />
```

Теперь можно попробовать запустить приложение, чтобы заставить фреймворк EF автоматически создать базу данных и таблицу. Добавьте несколько записей в таблицу `Guitars` и отпразднуйте это! Как должна выглядеть веб-страница после ввода адреса http://localhost:*[port]*/Guitars в браузере, показано на рис. 1.4.

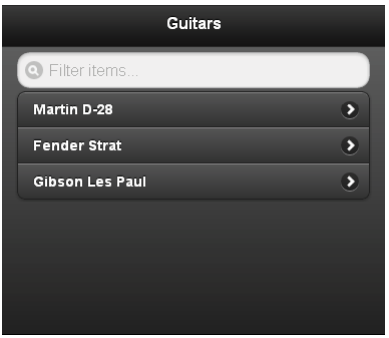

Рис. 1.4. Список гитар, полученный с помощью ASP.NET MVC 4 и jQuery Mobile

Извлечение данных

Новый синтаксис запросов, который был показан в предыдущем разделе, выглядит и действует подобно LINQ в C#/VB.NET. Ниже представлено несколько коротких примеров его использования:

Извлечение информации о гитаре по названию.

```
member x.GetByName name =
     use context = new FsMvcAppEntities()
     query { for g in context.Guitars do
             where (g.Name = name) }
     |> Seq.toList
```

Сортировка списка гитар по названию.

```
member x.GetAllAlphabetic () =
     use context = new FsMvcAppEntities()
     query { for g in context.Guitars do
             sortBy g.Name }
     |> Seq.toList
```

Извлечение первых X записей.

```
member x.GetTop rowCount =
     use context = new FsMvcAppEntities()
     query { for g in context.Guitars do
             take rowCount }
     |> Seq.toList
```

Дополнительные примеры можно найти на странице http://fsharp3sample.codeplex.com/.

> **Примечание.** Во многих примерах выше можно использовать F# PowerPack Linq, хотя синтаксис этого инструмента не отличается той ясностью.

Извлечение данных с использованием поставщиков типов

В версии F# 3.0 появилась еще одна новая особенность – *поставщики типов* (type providers), еще больше упрощающая взаимодействия с базами данных. Чтобы воспользоваться поставщиком типов для доступа к базе данных, сначала необходимо добавить ссылку на сборку `FSharp.Data.TypeProviders`. После этого появится возможность использовать стандартные поставщики типов для работы с базами данных, такие как `SqlDataConnection`. Этот поставщик извлекает схему базы данных и генерирует соответствующие типы. Например:

```
open Microsoft.FSharp.Data.TypeProviders

type DbConnection =
     SqlDataConnection<ConnectionStringName="FsMvcAppExample",
                       ConfigFile="web.config">

type GuitarsRepository2() =
     member x.GetAll () =
          use context = DbConnection.GetDataContext()
          query { for g in context.Guitars do
                  select (Guitar(Id = g.Id, Name = g.Name)) }
          |> Seq.toList
```

Примечание. Чтобы научить механизм IntelliSense распознавать свойства и методы `context` внутри проекта F# WebApp, необходимо приложить дополнительные усилия. Для этого просто создайте файл web.config в проекте F# WebApp и добавьте соответствующие элементы в строку подключения кс базе данных.

Поставщик типов позволяет использовать тот же синтаксис запросов, что был показан в разделе «Entity Framework» (выше). Возможно, не все поймут, чем это решение отличается от предыдущего, на основе фреймворка Entity Framework. Самое важное достижение состоит в том, что тип `FsMvcAppEntities` стал ненужным и от него можно полностью отказаться. Кроме того, класс модели `Guitar` стал проще, благодаря устранению атрибута `[<Key>]`.

Мы не будем погружаться в тонкости работы поставщиков типов, так как это выходит за рамки книги, но в общих чертах их работу можно описать так:

1. Добрая фея вдувает в ваш компьютер волшебную пыль.
2. И теперь вы можете использовать вычислительные выражения запросов для взаимодействия с базами данных.

Примечание. Желающие поближе познакомиться с внутренним устройством поставщиков типов, могут обратиться к документации на сайте MSDN (http://msdn.microsoft.com/ru-ru/library/hh361034.aspx), описывающей создание поставщиков типов. Кроме того, я написал пример реализации собственного поставщика типов, который можно найти в моем блоге (http://bit.ly/dmohltypeproviderexample). Имейте в виду, что этот пример написан с использованием предварительной версии F# 3.0 для разработчиков. Синтаксис F# мог претерпеть некоторые изменения с тех пор.

К счастью для нас, специфика работы этого механизма не имеет большого значения. Достаточно знать только синтаксис, с которым мы только что познакомились. Стандартные поставщики типов предоставляют множество преимуществ, если база данных, с которой вы взаимодействуете, уже существует. Однако, последующие примеры в этой главе используют подход «сначала код». Поэтому в них я буду использовать подход на основе фреймворка Entity Framework, применявшийся выше. В главе 2 я покажу пример на основе поставщиков типов.

Использование преимуществ F#

Теперь у вас есть все необходимое для создания простого веб-приложения на основе шаблона F#/C# ASP.NET MVC 4. Вы также познакомились с некоторыми замечательными особенностями языка F#, такими как типы записей, выражения последовательностей, прямой конвейерный оператор и одно из стандартных вычислительных выражений. Однако это лишь незначительная часть возможностей языка F#. Следующие несколько разделов познакомят вас с другими «волшебными» особенностями F#, которые помогут вам на вашем пути.

Первое, что можно было заметить в предыдущих примерах, – мы писали код для работы с ASP.NET MVC в объектно-ориентированном стиле. В этом нет ничего плохого и такой стиль с успехом можно использовать для создания надежных веб-приложений; однако, есть некоторые преимущества, которые можно получить с переходом на более функциональную парадигму программирования.

Переход на функциональную парадигму

Возможно, вы заметили пару важных особенностей в контроллере и репозитории. Репозиторий реализован в виде отдельного класса `GuitarRepository`. Если в будущем потребуется другой репозиторий, например, для хранения информации о трубах, Вам придется определить новый, очень похожий класс `TrumpetRepository`. Такая необходимость часто приводит к созданию массы кода, нарушающего принцип, который гласит: «не повторяйся» (Don't Repeat Yourself, DRY). Один из способов решения этой проблемы, основанный на функциональной парадигме, заключается в создании универсального модуля с функциями, принимающими другие функции в виде аргументов.

Функции, принимающие и возвращающие другие функции, называются *функциями высшего порядка* (higher-order functions). Следующий пример:

```
namespace FsWeb.Repositories

open System
open System.Linq

module Repository =
    let get (source:IQueryable<_>) queryFn =
        queryFn source |> Seq.toList

    let getAll () =
        fun s -> query { for x in s do
                         select x }
```

определяет модуль `Repository` – именованную группу конструкций на языке F#, таких как функции. Функции в модуле `Repository` являются полностью универсальными и обеспечивают возможность многократного использования, благодаря тому, что F# интерпретирует функции, как сущности первого рода (first-class citizen). Функция `get` принимает два аргумента.

- Объект `source` типа `IQueryable<_>`;
- Функцию, принимающую объект `source` как аргумент. Это может быть любая функция, принимающая объект `source` в последнем аргументе и возвращающая результат, который можно передать методу `Seq.toList`, выполняющему вычислительное выражение запроса.

> **Примечание.** Концептуально модуль Repository можно сравнить с шаблоном универсального репозитория на C#, использующим `IRepository<T>` и/или `Repository<T>`. Некоторые считают это ненужным слоем абстракции, но на мой взгляд такой подход улучшает читаемость кода и устраняет необходимость его повторения. Некоторые примеры в этой книге используют прием на основе универсального репозитория, но сами вычислительные выражения запросов могут использоваться без дополнительных абстракций.

Наличие функции `get` позволяет создавать дополнительные функции запросов, которые можно передавать функции `get` во втором аргументе. Примером таких функций является функция `getAll`. Ниже приводятся другие примеры подобных функций:

```
let find filterPredFn =
    filterPredFn
    |> fun fn s -> query { for x in s do
                           where (fn()) }

let getTop rowCount =
    rowCount
    |> fun cnt s -> query { for x in s do
                            take cnt }
```

> **Примечание.** На языке C# нечто подобное можно реализовать в виде типа `Func<T, TResult>`, но синтаксис такой реализации очень быстро может уйти далеко от идеала.

Чтобы воспользоваться преимуществами этого универсального репозитория, необходимо внести несколько изменений в код контроллера. Самое интересное, что эти изменения повышают удобство тестирования. Прежде, чем я покажу, как внести эти изменения в контроллер, я должен познакомит вас с некоторыми дополнительными понятиями функционального программирования.

Конвейеры и частичное применение функций

Прием композиции функций является одним из ключевых в арсенале разработчиков на F#. Говоря простым языком, композиция – это объединение в цепочку небольших функций с узкой областью применения, с целью реализации более сложных и многомерных процессов и алгоритмов. Этот прием позволяет создавать приложения для решения сложнейших задач, снижать вероятность появления ошибок, уменьшать трудозатраты на сопровождение и увеличивать удобочитаемость кода.

Вы уже видели несколько примеров композиции функций в виде применения прямого конвейерного оператора – одной из реализаций этой концепции. Прямой конвейерный оператор вызывает первую функцию или метод в цепочке и передает результат вызова в виде аргумента следующей функции или методу. Этот порядок действий иллюстрирует следующий простой пример:

```
let function1 () =
     "Hello "
```

```
let function2 firstString =
     firstString + "Daniel "
let function3 combinedString =
     combinedString + "Mohl"

let result = function1() |> function2 |> function3
printfn "%s" result
```

В этом примере определяется три функции с незамысловатыми именами `function1`, `function2` и `function3`. Далее вызывается каждая из этих функций и окончательный результат присваивается переменной `result`. В самом конце значение `result` выводится на экран. Самое интересное здесь заключается в цепочке вызовов функций – результат вызова функции `function1` передается функции `function2` в виде аргумента. Результат вызова функции `function2` передается функции `function3`. В конечном счете получается строка «Hello Daniel Mohl», которая затем выводится на экран.

Кому-то все это может показаться элементарным, особенно имеющим опыт разработки программ на F#, но для нас очень важно разобраться с основными приемами, прежде чем переходить к более сложным понятиям.

> **Примечание.** В языке F# имеются и другие операторы, обеспечивающие возможность композиции функций. Более подробную информацию об этих операторах, таких как `||>`, `<||`, `<|`, `>>` и `<<` можно найти на странице http://bit.ly/fssymbolsandoperators[1]. Некоторые из них мы будем обсуждать здесь и в других главах этой книги.

Следующее понятие, имеющее отношение к композиции функций и которое необходимо усвоить, прежде чем двигаться дальше, – это *частичное применение функций* (partial function application). Язык F# поддерживает возможность частичного применения функций – известную также как каррирование (curried functions) – которое осуществляется простым разделением аргументов пробелами. Например:

```
let function1 firstString secondString thirdString =
    "Hello " + firstString + secondString + thirdString
```

[1] Существует аналогичная страница на русском языке, но она содержит несколько устаревшую информацию (http://msdn.microsoft.com/ru-ru/library/vstudio/dd233228(v=vs.100).aspx). – *Прим. перев.*

Эта функция позволяет передать лишь часть аргументов и в этом случае она вернет новую функцию, принимающую недостающие аргументы, не включенные в первый вызов.

Например, вызов функции `partiallyAppliedFunc` в следующем фрагменте вернет новую функцию, ожидающую получить единственный строковый аргумент:

```
let partiallyAppliedFunc = function1 "Daniel" "Mohl"
```

Теперь можно попробовать использовать эти концепции, чтобы в реализации `GuitarsController` использовать более функциональный подход.

Создание более функционального контроллера

Используя приемы композиции и частичного применения функций, мы можем изменить реализацию `GuitarsController`, чтобы задействовать в нем новый модуль `Repository`. Сначала я покажу полную реализацию контроллера, а потом мы разберем ее по частям:

```
namespace FsWeb.Controllers

open System
open System.Web.Mvc
open FsWeb.Models
open FsWeb.Repositories
open Repository

[<HandleError>]
type GuitarsController(context:IDisposable, ?repository) =
     inherit Controller()

     let fromRepository =
          match repository with
          | Some v -> v
          | _ -> (context :?> FsMvcAppEntities).Guitars
                 |> Repository.get

     new() = new GuitarsController(new FsMvcAppEntities())

     member this.Index () =
```

```
        getAll() |> fromRepository
        |> this.View

    override x.Dispose disposing =
        context.Dispose()
        base.Dispose disposing
```

Первое изменение заключается в простом открытии модуля `Repository` подобно тому, как открывается любое другое пространство имен. В действительности этот шаг необходим, только чтобы повысить удобочитаемость кода. Следующее изменение касается главного конструктора – теперь он принимает объект, реализующий интерфейс `IDisposable` и необязательный аргумент с репозиторием. Самое большое отличие в том, что теперь репозиторий является функцией, а не объектом. Важную роль здесь играет механизм определения типов, идентифицирующий сигнатуру функции, исходя из особенностей ее использования. Это позволяет сохранить код кратким и ясным. Использование функции вместо объекта дает возможность передать в качестве репозитория любую функцию с соответствующей сигнатурой, что существенно упрощает тестирование, которое может доставлять головную боль при работе с фреймворком Entity Framework.

Следующее изменение, на которое следует обратить внимание, – определение `fromRepository`. Этот фрагмент кода проверяет входной параметр `repository`. Если в этом параметре ничего не было передано (что является нормальным ходом выполнения), выполняются подготовительные операции по созданию функции, которая затем будет использоваться для извлечения данных. Это – наглядный пример практического применения приемов композиции функций посредством прямого конвейерного оператора и частичного применения каррированной функции (curried function). Прямой конвейерный оператор передает значение `context.Guitars` функции `Repository.get`. В результате `Repository.get` возвращает частично примененную (partially applied) функцию с фиксированным первым аргументом (значением `context.Guitars`), которую позднее можно будет вызвать с единственным аргументом.

Примечание. Вызов `Repository.get` можно было бы записать как `get`, потому что выше мы уже открыли модуль `Repository`, но я считаю, что использование более длинного имени улучшает читаемость кода.

Далее следует конструктор без параметров, создающий новый экземпляр `FsMvcAppEntities` и передающий его главному конструктору. Репозиторий в этом случае не передается. Это дает нам дополнительные выгоды, потому что внутри веб-приложения желательно использовать реализацию репозитория по умолчанию.

Последнее интересное изменение заключается в использовании для извлечения данных функции `fromRepository`, созданной в главном конструкторе, и функции `getAll` из модуля `Repository`. Их результаты передаются представлению точно так же, как в предыдущем примере. Как показано в этом примере, язык F# дает еще одно любопытное преимущество – гибкость языка может сделать ваш код более читаемым. Выражение `getAll |> fromRepository` читается практически так же, как если бы я написал или сказал это на обычном английском. Это выражение можно также записать как `fromRepository (getAll())` или `fromRepository <| getAll()`, но в этом случае читать его становится сложнее. F# дает массу вариантов, из которых можно выбрать лучший для решения поставленной задачи.

Упрощение за счет сопоставления с образцом

Теперь у вас есть удобочитаемая и простая в сопровождении реализация контроллера и репозитория, написанная в более функциональном стиле, но F# имеет множество других особенностей, несущих дополнительные выгоды. В этом разделе я покажу лишь самую малую часть возможностей механизма сопоставления с образцом в языке F#.

Механизм *сопоставления с образцом* (pattern matching) – это одна из особенностей, позволяющих управлять ходом выполнения программы и/или преобразованием данных на основе определяемых вами правил, или образцов. В F# поддерживается большое разнообразие типов образцов. Полный их перечень можно найти по адресу: http://msdn.microsoft.com/ru-ru/library/vstudio/dd547125(v=vs.100).aspx.

Чтобы объяснить, как сопоставление с образцом может помочь в проектах ASP.NET MVC, я проведу вас через процесс создания новой страницы. Эта страница помогает создать новую запись `Guitar`. Для этого вам необходимо добавить новое представление ASP.NET MVC и несколько новых методов контроллера. В примере ниже приводится разметка для нового представления, содержащаяся в новом файле с именем *Create.cshtml*:

```
@model FsWeb.Models.Guitar
<!DOCTYPE html>
<html>
<head>
     <title>Create a guitar</title>
     <meta name="viewport" content="width=device-width, initial-scale=1" />
     <link rel="stylesheet"
          href=
          "http://code.jquery.com/mobile/1.0.1/jquery.mobile-1.0.1.min.css" />
     <script src="http://code.jquery.com/jquery-1.6.4.min.js">
     </script>
     <script src=
          "http://code.jquery.com/mobile/1.0.1/jquery.mobile-1.0.1.min.js">
     </script>
</head>
<body>
     <div data-role="page" data-theme="a" id="guitarsCreatePage">
          <div data-role="header">
               <h1>Guitars</h1>
          </div>
          <div data-role="content">
               @using (Html.BeginForm("Create", "Guitars", FormMethod.Post))
               {
                    <div data-role="fieldcontain">
                         @Html.LabelFor(model => model.Name)
                         @Html.EditorFor(model => model.Name)
                         <div>
                              @Html.ValidationMessageFor(m => m.Name)
                         </div>
                    </div>
                    <div>
                         <button type="submit" data-role="button">Create</button>
                    </div>
               }
          </div>
     </div>
</body>
</html>
```

Сохраните этот файл в папке *Views\Guitars* проекта веб-приложения на C#. Как и предыдущий пример представления для ASP.NET, эта разметка приводится здесь исключительно в демонстрационных целях и не должна рассматриваться как нечто, «пригодное для использования в промышленных условиях».

Теперь необходимо добавить два новых метода в класс GuitarsController. Это место, где волшебство сопоставления с образцом вступает в игру. Новые методы представлены в примере ниже. Выделенный фрагмент кода – это выражение сопоставления с образцом:

```
[<HttpGet>]
member this.Create () =
     this.View()
[<HttpPost>]
member this.Create (guitar : Guitar) : ActionResult =
     match base.ModelState.IsValid with
     | false ->
          upcast this.View guitar
     // ... код, сохраняющий данные, которые будут добавляться позднее
     | true -> upcast base.RedirectToAction("Index")
```

Сосредоточим наше внимание на втором методе. Этот метод обрабатывает POST-запрос, включающий информацию, необходимую для создания записи. Самое интересное здесь – это выражение сопоставления с образцом, определяющее как будет протекать выполнение в зависимости от результатов проверки класса переданной модели, в данном случае – guitar. Выражение сопоставления с образцом в данном примере фактически является аналогом инструкции switch или case. (Это простое выражение сопоставления можно было бы записать в виде выражения if...then...else. Подробнее о таких выражениях можно узнать по адресу: http://msdn.microsoft.com/ru-ru/library/vstudio/dd233231(v=vs.100).aspx.)

> **Примечание.** Возможно вы заметили, что результат HttpPost-версии метода Create требует приведения к типу ActionResult. Вы можете повысить удобочитаемость кода, добавив простую функцию, выполняющую приведение типа:
>
> ```
> let asActionResult result = result :> ActionResult
> ```
>
> которую затем можно использовать, как показано ниже:
>
> ```
> guitar |> this.View |> asActionResult
> ```

Все это хорошо, но как быть, если потребуется проверить несколько элементов, например, тип модели и некоторого свойства этой модели? Например, наряду с типом модели может потребоваться убедиться в отсутствии слова «broken» (требует ремонта) в свойстве guitar.Name. Единственный способ решить эту задачу с помощью ин-

струкции switch – реализовать вложенные проверки. Но при таком подходе код легко может выйти из-под контроля, а его сопровождение превратится в сущий кошмар.

Оператор сопоставления с образцом в языке F# легко справляется с этой проблемой, как показано ниже:

```
let isNameValid = Utils.NullCheck(guitar.Name).IsSome &&
                  not (guitar.Name.Contains("broken"))
match base.ModelState.IsValid, isNameValid with
| false, false | true, false | false, true ->
     upcast this.View guitar
| _ -> upcast base.RedirectToAction("Index")
```

Этим примером я хотел показать, что выражение сопоставления с образцом – это намного больше, чем простая инструкция switch. Первая строка теперь определяет кортеж (tuple), в первом значении которого передается результат вызова base.ModelState.IsValid, а во втором – результат isNameValid. Теперь мы можем выполнить сопоставление этого кортежа с образцом.

> **Примечание.** Вызов Utils.NullCheck(guitar.Name).IsSome в предыдущем примере определяет, не является ли переданный объект guitar значением null. Он возвращает значение Some типа Option, если объект guitar не равен null, и None – в противном случае. Ниже приводится функция Utils.NullCheck, реализующая эту проверку и представляющая еще один пример практического применения сопоставления с образцом с использованием несколько иного синтаксиса, чем вы видели до сих пор:
>
> ```
> let NullCheck = function
>     | v when v <> null -> Some v
>     | _ -> None
> ```

Первый образец в примере выше сопоставляется с кортежем, описанным выше. Кроме того, он реализует логику OR (ИЛИ), что делает код более кратким. Сопоставление считается успешным, если выражению был передан кортеж (false, false), (true, false) или (false, true).

> **Примечание.** Проверка свойства Name в этом примере используется только с целью показать некоторые дополнительные возможности механизма сопоставления с образом в языке F#. В настоящем приложении подобные проверки предпочтительнее организовывать с помощью собственного атрибута проверки, помещаемого в модель и, возможно, выполняющего проверку не только на стороне сервера, но и на стороне клиента.

Механизм сопоставления с образцом обладает массой других замечательных особенностей, включая возможность связывать входные данные со значениями для преобразования, реализовывать более сложную логику проверки данных и выполнять сопоставление записей.

Примечание. Возможность сопоставления с записями – одна из причин, объясняющих, почему следует отдавать предпочтение записям, а не классам, если это возможно. В числе других причин: неизменяемость, компактность и простота создания новых записей, отличающихся от оригинала.

Дополнительные темы и понятия

Я показал несколько способов, как начать пользоваться преимуществами языка F# в приложениях на основе ASP.NET MVC. Теперь ваши контроллеры и модели будут получаться компактными, читаемыми и более функциональными по своей природе. Некоторые из преимуществ, показанных выше, обеспечивают малозаметные улучшения. Другие существенно улучшают читаемость кода и надежность приложения. Однако вы, возможно, еще не осознали, насколько F# позволяет упростить сопровождение кода. Выше я привел один из примеров преобразования кода, чтобы показать, как сделать приложение более функциональным. В течение этого процесса я внес несколько изменений в типы, связанные с различными методами и функциями. Благодаря механизму определения типов лишь в одном месте мне пришлось изменить сигнатуру, а остальная часть кода автоматически восприняла изменения. Для кого-то это может оказаться большой победой!

В следующем разделе я познакомлю вас с более сложными понятиями, не имеющими прямого отношения к решениям на основе ASP.NET MVC, но часто используемых совместно с ними.

Улучшение времени отклика с помощью асинхронных операций

Давно известно, что асинхронные потоки операций являются ключом к увеличению отзывчивости веб-сайтов. F# поддерживает особенность с названием *асинхронные потоки операций* (asynchronous workflows, или async workflows), которая серьезно упрощает выполнение асинхронных вызовов к серверу. Ниже показан базовый синтаксис:

```
async {
     let client = new WebClient()

     return! client.AsyncDownloadString(Uri("http://www.yahoo.com"))
}
```

Проницательный читатель заметит, что синтаксис оператора `async` несколько напоминает синтаксис запросов. Это обусловлено тем, что `async` – это такое же вычислительное выражение, как и запросы. Вычислительные выражения могут принимать массу обличий.

> **Примечание.** Оператор `!` (произносится как «банг») в предыдущем примере сообщает вычислительному выражению, что эта строка будет выполнять некоторые дополнительные операции, помимо тех, что предполагаются базовой реализацией, вызывая определенные функции (в данном случае `Bind` и `Return`). Пример создания собственного вычислительного выражения будет представлен ниже в этой главе.

Этот простой пример асинхронного потока операций создает новый экземпляр `WebClient` в блоке `async`. Затем он загружает содержимое из указанного URI, не блокируя выполнение основного потока выполнения. Следует заметить, что блок `async` возвращает генератор задания, которое не будет запущено, пока это не будет сделано явно. Запустить предыдущий фрагмент, например, можно вызовом `Async.Start`.

> **Примечание.** Томас Петричек (Tomas Petricek) написал замечательную серию статей (http://tomasp.net/blog/async-csharp-differences.aspx), где описал различия между механизмами асинхронных операций в F# и C# 5.0. Вам стоит это почитать!

Как можно использовать эту особенность в проектах на основе ASP.NET MVC? Для начала ее можно применить для организации внешних вызовов из контроллеров. Но чаще она используется для создания асинхронных контроллеров. Томас Петричек (Tomas Petricek) и Джон Скит (Jon Skeet) представили пример такого контроллера (http://msdn.microsoft.com/en-us/library/hh304367.aspx). Другим примером является использование асинхронных потоков операций в сочетании с объектами `MailboxProcessor` для создания легковесных процессов, которые могут применяться для решения самых разных задач, о чем и рассказывается в следующем разделе.

Кеширование с применением MailboxProcessor

Своим действием класс `MailboxProcessor` в языке F# напоминает настоящее волшебство. Чем ближе вы будете узнавать его, тем больше различных применений ему вы сможете найти. `MailboxProcessor` — часто используется под псевдонимом `agent` — благодаря использованию механизма асинхронных потоков операций он позволяет запускать десятки тысяч изолированных процессов для решения ваших задач. Это означает, что вы можете распределить работу между несколькими экземплярами `MailboxProcessor`, по аналогии с потоками выполнения, без лишних накладных расходов, необходимых для запуска каждого нового потока выполнения. Кроме того, объекты `MailboxProcessor` в основном обмениваются информацией посредством сообщений (хотя это не единственный способ), что позволяет избавиться от проблем, свойственных использованию разделяемого состояния в многопоточной среде.

Для решения поставленных задач, каждый экземпляр `MailboxProcessor` имеет виртуальную «очередь сообщений», которая постоянно следит за прибытием новых сообщений. После поступления сообщения, его можно извлечь из очереди и передать для обработки. Для определения типа прибывшего сообщения, чтобы выбрать соответствующую процедуру обработки, обычно используется операция сопоставления с образцом. После обработки сообщения отправителю может быть послан ответ (синхронно или асинхронно).

Следующий пример демонстрирует использование `MailboxProcessor` для хранения кешированных данных:

```
namespace FsWeb.Repositories

module CacheAgent =
    // Размеченное объединение допустимых типов сообщений
    type private Message =
        | Get of string * AsyncReplyChannel<obj option>
        | Set of string * obj

    // Основной агент
    let private agent = MailboxProcessor.Start(fun inbox ->
        let rec loop(cacheMap:Map<string, obj>) =
            async {
                let! message = inbox.Receive()
```

```
                match message with
                | Get(key, replyChannel) ->
                    Map.tryFind key cacheMap |> replyChannel.Reply
                | Set(key, data) ->
                    do! loop( (key, data) |> cacheMap.Add)
                do! loop cacheMap
            }
        loop Map.empty)

// Общедоступная функция, извлекающая данные из кеша в виде значений
// типа Option
let get<'a> key =
    agent.PostAndReply(fun reply -> Message.Get(key, reply))
    |> function
        | Some v -> v :?> 'a |> Some
        | None -> None

// Общедоступная функция, сохраняющая данные в кеше
let set key value =
    Message.Set(key, value) |> agent.Post
```

На первый взгляд этот пример может выглядеть пугающе, но дополнительные пояснения помогут расставить все по своим местам.

Сообщения, как значения типа размеченного объединения

В первую очередь в модуле `CacheAgent` объявляется тип с именем `Message`, определяющий все допустимые сообщения. Для объявления этого типа была использована особенность языка F#, которая называется *размеченные объединения* (discriminated unions), позволяющая перечислять группы родственных типов и значений. Она особенно удобна в данном конкретном случае, так как позволяет определить контракты сообщений для обработки агентом. Размеченные объединения также дают возможность определять различные сигнатуры для каждого объявляемого типа, что существенно расширяет область их применения. Размеченные объединения могут применяться в самых разных ситуациях, и на протяжении книги я еще на раз буду давать примеры их использования.

В предыдущем примере размеченное объединение `Message` определяет только тип входных сообщений. Однако при необходимости в него легко можно добавить определения типов выходных сообщений, как показано в следующем примере:

```
type private MessageExample2 =
     | Get of string * AsyncReplyChannel<Reply>
     | Set of string * obj
and Reply =
     | Failure of string
     | Data of obj
```

Основной агент

Непосредственно за объявлением типа `Message` следует определение основного агента. Он создает и тут же запускает новый экземпляр `MailboxProcessor`, который непрерывно проверяет поступление новых сообщений. Ему передается анонимная функция, содержащая определение рекурсивной функции `loop`. Функция `loop` имеет единственный параметр с именем `cacheMap`, который повторно передается функции, чтобы обеспечить возможность управления состоянием. Использование асинхронного потока операций дает агенту возможность выполнять цикл проверки, не блокируя работу приложения. Именно внутри этого блока проверяется наличие новых сообщений в очереди.

При обнаружении нового сообщения, будет выполнено сопоставление с образцом, чтобы определить тип сообщения и выбрать соответствующую процедуру обработки. При получении сообщения типа `get`, будет предпринята попытка найти в `cacheMap` запись с указанным ключом. Если она увенчается успехом, найденное значение буде возвращено в виде экземпляра `Some(value)`, в противном случае будет возвращено значение `None`. В заключение полученный результат возвращается отправителю сообщения.

> **Примечание.** И снова мы видим пример использования типа `Option`. Применение типа `Option` для передачи значений, которые могут иметь значение `null`, увеличивает надежность приложения, так как помогает предотвратить ошибку разыменования пустого указателя. В F# вы часто будете видеть, что явные определения предпочтительнее неявных. Тип `Option` позволяет явно определить, какие значения могут присваиваться, а какие нет, тогда как значение `null` может соответствовать и недопустимому значению, и значению, соответствующему состоянию по умолчанию.

При получении сообщения типа `set`, заданная пара ключ/значение будет добавлена в `cacheMap` и новое значение `cacheMap` будет передано в виде аргумента в рекурсивный вызов функции `loop`.

В конце выполняется запуск рекурсивного цикла с пустым ассоциативным массивом `Map`.

Примечание. Значение `cacheMap` в примере использования `MailboxProcessor` имеет тип `Map` – тип неизменяемой структуры данных, обладающий базовой функциональностью ассоциативных массивов, или словарей.

В оставшейся части примера определяется общедоступный API для взаимодействия с агентом. Это обычный фасад перед API агента, несколько упрощающий работу с ним.

Примечание. Дополнительные сведения и примеры использования `MailboxProcessor` можно найти по адресу: http://msdn.microsoft.com/ru-ru/library/ee370357.aspx.

Использование агента CacheAgent

Пользоваться новым агентом `CacheAgent` весьма просто. Достаточно лишь вызывать общедоступные функции `get` и `set`. Эти вызовы легко можно добавить в действия любого контроллера, но лучше добавить их в реализацию модуля `Repository` и предоставить возможность включения и выключения механизма кеширования. Ниже приводится измененная версия модуля `Repository`, где новый код выделен жирным шрифтом:

```
module Repository =
    let withCacheKeyOf key = Some key

    let doNotUseCache = None

    let get (source:IQueryable<_>) queryFn cacheKey =
        let cacheResult =
            match cacheKey with
            | Some key -> CacheAgent.get<'a list> key
            | None -> None

        match cacheResult, cacheKey with
        | Some result, _ -> result
        | None, Some cacheKey ->
                let result = queryFn source |> Seq.toList
                CacheAgent.set cacheKey result
                result
        | _, _ -> queryFn source |> Seq.toList

    let getAll () =
        fun s -> query { for x in s do
```

```
                            select x }

    let find filterPredFn =
        filterPredFn
        |> fun fn s -> query { for x in s do
                               where (fn()) }

    let getTop rowCount =
        rowCount
        |> fun cnt s -> query { for x in s do
                                take cnt }
```

Первые две функции определяют удобный способ включения и выключения механизма кеширования. При включении механизма кеширования указывается ключевая строка. Ниже приводится несколько примеров использования этих функций:

```
let top2Guitars = getTop 2 |> fromRepository <| doNotUseCache

getAll() |> fromRepository <| withCacheKeyOf("AllGuitars") |> this.View
```

> **Примечание.** Обратный конвейерный оператор (backward pipe) (`<|`) в предыдущем примере обеспечивает передачу результата выражения справа от него функции слева.

Измененная версия функции `get` может извлекать значения из кеша (если это необходимо и эти значения имеются в кеше), сохранять в кеше значения, извлеченные из базы данных, для ускорения обработки последующих запросов, или всегда извлекать их из базы данных. Первое выражение сопоставления с первым образцом выясняет, включен ли режим кеширования. Если условие `Some key` выполняется, производится попытка извлечь значение из кеша `CacheAgent`. Результат попытки связывается с переменной `acheResult`. Если кеширование отключено, с переменной связывается значение `None`.

Следующее выражение сопоставления с образцом проверяет три возможных варианта развития событий.

- Из кеша было извлечено некоторое значение и в этом случае оно возвращается вызывающей программе.
- Значение не было найдено в кеше `CacheAgent`, но кеширование включено. В этом случае выполняется запрос к базе данных, результат сохраняется в кеше `CacheAgent` и возвращается вызывающей программе.

- Кеширование отключено. В этом случае все запросы будут выполняться через базу данных, а извлеченные значения не будут сохраняться в кеше.

Описанию примеров использования `MailboxProcessor` можно было бы посвятить целую книгу. Данный пример реализации кеширования легко можно было бы дополнить такими особенностями, как предельное время хранения данных в кеше, автоматический переход на использование другого источника данных в случае отказа, автоматическое обновление кешируемых данных, распределенная обработка, рассылка событий при изменении данных, очистка кеша и так далее. В качестве дополнительных примеров использования `MailboxProcessor` можно назвать: любую фоновую обработку данных, для осуществления которой в других условиях вы могли бы задействовать фоновые потоки выполнения; демоны; архитектуры типа CQRS; механизмы извещений и серверы веб-сокетов.

Шина сообщений

После знакомства с `MailboxProcessor` вы уже знаете, с чего начать, когда в приложении на основе ASP.NET MVC потребуется организовать прием и передачу сообщений на F#. В этом разделе я подробнее остановлюсь на этой концепции и расскажу, как на F# реализовать шину сообщений (message bus). Шина сообщений дает массу преимуществ, от возможности писать масштабируемые и слабо связанные системы, до организации взаимодействий между различными платформами. В архитектурах, базирующихся на обмене сообщениями и использующих шины сообщений, основное внимание уделяется обобщенным контрактам обработки сообщений и их передаче. Звучит знакомо, не правда ли? В этом нет ничего удивительного, потому что `MailboxProcessor` фактически делает то же самое. В этой главе не так много места, чтобы охватить эту тему во всех подробностях, тем не менее, примеры в последующих разделах смогут послужить для вас неплохой отправной точкой.

Для простоты я создал небольшую библиотеку `SimpleBus`, на которую буду ссылаться на протяжении всей оставшейся части главы. Следует отметить, что хотя библиотека `SimpleBus` прекрасно подходит для примеров в этой книге, в ней отсутствуют некоторые важные особенности, такие как полноценный механизм публикации/подписки на события, транзакционные очереди, механизм запрос/ответ, поддержка взаимосвязанных сообщений, возможность передачи

сообщений различных типов через одну очередь, и многие другие. Для действующих приложений я рекомендую использовать одну из многочисленных реализаций шин сообщений, которые легко можно найти в Интернете. Несмотря на то, что библиотека SimpleBus требует дальнейшего расширения, чтобы ее можно было использовать в промышленном окружении, сам код на F#, демонстрируемый ниже, и описываемые понятия могут использоваться в действующих приложениях без каких-либо ограничений.

SimpleBus

Главная цель библиотеки SimpleBus – обеспечить возможность публикации сообщений для указанной конечной точки MSMQ (очереди сообщений Microsoft – Microsoft Message Queue), которые затем могут извлекаться другими процессами. Это закладывает основу для высоко масштабируемых, слабо связанных систем, деятельность которых основана на обмене сообщениями. Чтобы обеспечить возможность публикации сообщений, я определил функцию с именем publish, принимающую имя очереди queueName и сообщение message для сериализации (с помощью BinaryMessageFormatter) и передачи в очередь:

```
let publish queueName message =
    use queue = new MessageQueue(parseQueueName queueName)
    new Message(message, new BinaryMessageFormatter())
    |> queue.Send
```

Функция сначала создает новый экземпляр MessageQueue, позаботившись о том, чтобы память, выделенная для экземпляра MessageQueue, автоматически освобождалась, как только он станет не нужен. Желаемое имя очереди передается вспомогательной функции parseQueueName, описываемой далее в этом разделе. Результат вызова parseQueueName затем передается конструктору MessageQueue. После этого создается экземпляр типа Message (определен в пространстве имен System.Messaging), где сохраняется обобщенный объект сообщения, переданный в аргументе message. Конструктору Message также передается новый экземпляр BinaryMessageFormatter для сериализации сообщения в двоичное представление. Если этот аргумент опустить, по умолчанию будет использоваться XmlMessageFormatter. Я задействовал BinaryMessageFormatter по той простой причине, что этот класс обеспечивает более высокую производительность и позволяет отправлять в качестве сообщений простые записи F#. Далее но-

вый экземпляр Message передается методу Send экземпляра очереди сообщений MessageQueue.

Ниже приводятся две вспомогательные функции, используемые в функции publish:

```
let private createQueueIfMissing (queueName:string) =
     if not (MessageQueue.Exists queueName) then
          MessageQueue.Create queueName |> ignore

let private parseQueueName (queueName:string) =
     let fullName = match queueName.Contains("@") with
                    | true when queueName.Split('@').[1] <> "localhost" ->
                        queueName.Split('@').[1] + "\\private$\\" +
                             queueName.Split('@').[0]
                    | _ -> ".\\private$\\" + queueName
     createQueueIfMissing fullName
     fullName
```

Функция createQueueIfMissing проверяет наличие очереди и при ее отсутствии создает очередь, не поддерживающую транзакции. Механизм MSMQ позволяет публиковать сообщения не только в локальных но и в удаленных очередях. Тип очереди определяется ее именем. Функция parseQueueName просматривает переданное имя очереди, определяет, является ли она локальной, и форматирует имя очереди соответственным образом. Затем она вызывает createQueueIfMissing, чтобы гарантировать создание очереди, если это необходимо.

После определения функции publish и используемых ею вспомогательных функций, нам осталось лишь запустить процесс перекачки сообщений в очередь. Но, прежде чем я расскажу, как это сделать, я покажу и опишу функцию subscribe:

```
let subscribe<'a> queueName callback =
     let queue = new MessageQueue(parseQueueName queueName)

     queue.ReceiveCompleted.Add(
          fun (args) ->
               args.Message.Formatter <- new BinaryMessageFormatter()
               args.Message.Body :?> 'a |> callback
               queue.BeginReceive() |> ignore)

     queue.BeginReceive() |> ignore
     queue
```

Функция `subscribe` реализует возможность извлечения сообщений из очереди. Она поддерживает только один тип сообщений, который должен получаться после десериализации. Подобно функции `publish` она принимает имя очереди `queueName` в первом аргументе. Кроме того, она принимает функцию обратного вызова `callback` для обработки сообщений, извлекаемых из очереди.

Сначала `subscribe` создает новый экземпляр `MessageQueue`, практически так же, как это делает `publish`. Единственное существенное отличие в том, что связывание с переменной `queue` производится с помощью ключевого слова `let`, а не `use`. Это – важный момент, потому что означает, что функция не будет освобождать ресурсы автоматически, и вы должны будете освободить их явно, вызовом метода `Dispose()` очереди, возвращаемой функцией `subscribe`. Этот дополнительный этап освобождения ресурсов является обязательным, потому что событию `ReceiveCompleted` может потребоваться обратиться к очереди намного позже вызова `subscribe`.

Затем функция `subscribe` создает обработчик события `System.Messaging.MessageQueue.BeginReceive`, которое будет генерироваться при появлении нового сообщения в очереди. Этот обработчик устанавливает механизм десериализации сообщения. Приводит тело сообщения к требуемому типу и передает его функции обратного вызова `callback`. Затем вызывается метод `BeginReceive()`, чтобы продолжить мониторинг очереди. Значение `IAsyncResult`, возвращаемое методом `BeginReceive()`, не представляет интереса, поэтому оно просто игнорируется.

Примечание. Как уже упоминалось в начале раздела, библиотека `SimpleBus` не предназначена для промышленного использования. Например, она не поддерживает возможность подключения нескольких подписчиков к одной очереди. Я намеренно опустил эту возможность для простоты. Пример простой реализации шины сообщений на F#, пригодной для промышленного использования, можно найти по адресу: https://github.com/dmohl/FsBus.

Публикация сообщений

Разделение ответственности играет важную роль во всех аспектах разработки программного обеспечения, и решения на основе обмена сообщениями не являются исключением. Одним из типичных подходов к такому разделению является следование принципу *разделения ответственности на команды и запросы* (Command Query Responsibility Segregation, CQRS). Этот принцип требует явного

отделения команд (операций, вызывающих изменение состояния) от запросов (которые просто возвращают данные, используемые только для чтения). Этот подход отлично согласуется с концепциями, присущими функциональным языкам программирования, таким как F#.

> **Примечание.** Следует отметить, что использование архитектур на основе сообщений не обязывает соблюдать принцип CQRS или наоборот. Однако они часто сопровождают друг друга, и многие понятия, связанные с принципом CQRS и/или другими архитектурами, где используются шины сообщений, отлично дополняют многие базовые принципы языка F#.

Чтобы протестировать функцию `publish`, можно последовать принципу CQRS и создать тип, который будет использоваться как команда для создания новой записи `guitar`. Поскольку типы сообщений должны быть доступны как издателю сообщений, так и подписчику, их определения лучше поместить в отдельную сборку. Для данного примера я создал новую сборку с именем `Messages`. Запись `CreateGuitarCommand` для публикации в очереди имеет следующее определение:

```
namespace Messages

type CreateGuitarCommand = { Name : string }
```

Сообщения этого типа легко можно публиковать в очереди с помощью `SimpleBus.publish`. Чтобы проверить ее в работе, можно дополнить метод `Create` контроллера `GuitarsController`. Ниже показана измененная версия этого метода, где новый код выделен жирным шрифтом:

```
[<HttpPost>]
member this.Create (guitar : Guitar) =
     match base.ModelState.IsValid with
     | false -> guitar |> this.View |> asActionResult
     | true ->
          {Messages.CreateGuitarCommand.Name = guitar.Name}
          |> SimpleBus.publish "sample_queue"
          base.RedirectToAction("Index") |> asActionResult
```

> **Примечание.** Вам потребуется установить библиотеку MSMQ на своем компьютере. Кроме того, необходимо будет запустить Visual Studio в режиме администратора (хотя бы один раз), чтобы приложение смогло создать требуемую очередь.

Новый код просто создает новую запись CreateGuitarCommand, присваивает полю Name значение свойства guitar.Name и передает ее функции SimpleBus.publish. В данном конкретном случае выражение записи можно сократить до {Name = guitar.Name}, а обо всем остальном позаботится механизм определения типов. Однако на практике часто существует несколько команд с одинаковыми именами полей, поэтому лучше явно определять их типы. Например, мы могли бы дополнительно определить команду DeleteGuitarCommand, как показано ниже:

```
type DeleteGuitarCommand = { Name : string }
```

Извлечение сообщений

Операция извлечения сообщений из очереди ничуть не сложнее их публикации. Часто сообщения извлекаются и обрабатываются кодом, который разворачивается в виде служб Windows; однако сообщения точно так же могут извлекаться и обрабатываться веб-приложениями. Для простоты можно реализовать извлечение только что опубликованных сообщений в виде консольного приложения:

```
open Messages
open System

printfn "Waiting for a message"

let queueToDispose =
     SimpleBus.subscribe<CreateGuitarCommand> "sample_queue"
       (fun cmd ->
            printfn "A message for a new guitar named %s was consumed" cmd.Name)

printfn "Press any key to quite\r\n"
Console.ReadLine() |> ignore
queueToDispose.Dispose()
```

Это приложение подписывается на получение сообщений, которые после десериализации превращаются в записи CreateGuitarCommand. Функция, что передается во втором аргументе, функции subscribe, просто выводит некоторый текст, включающий название гитары, полученное из сообщения. Последняя строка в этом примере освобождает объект MessageQueue.

Стиль продолжений

Сейчас программа извлечения сообщений работает замечательно. Но, что случится, если возникнет ошибка, пока сообщение из очереди обрабатывается внутри функции `subscribe`? В данной реализации процесс извлечения сообщений просто остановится, потеряв сообщение и не известив никого о проблеме. Очевидно, что такое поведение нам не подходит. Нам необходимо, чтобы извлечение сообщений из очереди было продолжено, а приложение-подписчик было извещено о проблеме.

Один из способов решить поставленную задачу – использовать так называемый, *стиль продолжений* (Continuation-Passing Style, CPS). Стиль продолжений – это прием, когда функции (в частности действия контроллеров) вызывают другие функции по завершении, вместо того, чтобы просто вернуть управление вызывающей программе. Простейшим примером может служить функция обратного вызова, подобная той, что передается функции `subscribe` – фактически она является явно передаваемым продолжением.

Можно ли использовать этот прием для обработки ошибочных сообщений? Безусловно! Достаточно просто передать обработчик ошибок, который будет вызываться из функции `subscribe` при возникновении исключений, и приложение-потребитель будет извещаться обо всех проблемах и сможет реагировать соответственно. Для этого изменим функцию `subscribe`, как показано ниже (изменения выделены жирным):

```
let subscribe<'a> queueName success failure =
     let queue = new MessageQueue(parseQueueName queueName)

     queue.ReceiveCompleted.Add(
          fun (args) ->
               try
                    args.Message.Formatter <- new BinaryMessageFormatter()
                    args.Message.Body :?> 'a |> success
               with
               | ex -> failure ex args.Message.Body
               queue.BeginReceive() |> ignore)

     queue.BeginReceive() |> ignore
     queue
```

Теперь функция `subscribe` принимает две функции в аргументах. Первая будет вызываться в случае успеха, а вторая – в случае ошиб-

ки. Единственное, что нужно изменить в вызове функции `subscribe` — передать ей функцию, вызываемую в случае ошибки:

```
let queueToDispose =
     SimpleBus.subscribe<CreateGuitarCommand> "sample_queue"
       (fun cmd ->
            printfn "A message for a new guitar named %s was consumed" cmd.Name)
       (fun (ex:Exception) o ->
            printfn "An exception occurred with message %s" ex.Message)
```

В главе 2 я покажу дополнительные примеры реализации этого приема, включая возможность использования `Async.StartWithContinuations` для обработки успешного или ошибочного извлечения сообщений, а также для остановки асинхронных потоков операций.

Создание собственных вычислительных выражений

Выше в этой главе я показал несколько различных вычислительных выражений (computation expressions), поддерживаемых в языке F#. Стандартные вычислительные выражения обладают весьма широкими возможностями, но F# не ограничивается ими и позволяет вам создавать собственные вычислительные выражения. Однако, прежде чем погрузиться в их изучение, я хотел бы потратить совсем немного времени, чтобы дать максимально упрощенное определение.

В простейшем случае вычислительные выражения можно представить как обертки типов, обладающие своими последовательностями операций, которые могут применяться к заданному множеству кода на F#. Операции в последовательности применяются к различным точкам в процессе выполнения обернутого кода. Кроме того, конкретные операции внутри этой последовательности могут зависеть от особенностей вызова обернутого кода. Это позволяет определять небольшие строительные блоки, из которых затем можно будет составлять весьма сложные последовательности, но очень простые в использовании.

Чтобы показать, как сконструировать простое вычислительное выражение, я проведу вас через пример создания такого выражения с именем `publish`. Как можно догадаться из названия, это нестандартное вычислительное выражение выполняет публикацию сообщений в очередь подобно непосредственному вызову функции `publish` из библиотеки `SimpleBus`, обсуждавшейся в предыдущем разделе. Пре-

жде чем заняться реализацией этого вычислительного выражения, я покажу, на что будет похож синтаксис его использования:

```
publish {
     do! SendMessageWith("sample_queue",
            {Messages.CreateGuitarCommand.Name = guitar.Name})
}
```

Сами вычислительные выражения могут быть весьма и весьма сложными, но синтаксис их использования, как показывает предыдущий пример, остается очень простым. Ниже приводится реализация обсуждаемого вычислительного выражения:

```
module PublishMonad

// Определение размеченного объединения SendMessageWith
type SendMessageWith<'a> = SendMessageWith of string * 'a

// Определение типа построителя PublishBuilder
type PublishBuilder() =
     member x.Bind(SendMessageWith(q, msg):SendMessageWith<_>, fn) =
          SimpleBus.publish q msg
     member x.Return(_) = true

// Создание именованного построителя
let publish = new PublishBuilder()
```

Первое, на что следует обратить внимание – это определение типа `PublishBuilder`. Вы можете выбрать для своего типа любое имя, однако общепринято давать имена, как в этом примере, где *имя построителя* (builder name) (например, `publish`) записывается в стиле языка Pascal и к нему добавляется слово `Builder`. *Тип построителя* (builder type) (например, `PublishBuilder`) может объявлять различные методы, определяющие особенности работы построителя. Метод `Bind` вызывается, когда в вычислительном выражении используются символы `let!` или `do!` (произносятся как «лет-банг» и «ду-банг», соответственно). Метод `Return` является обязательным и вызывается практически всегда. Так как в данном случае не требуется никакой дополнительной логики, наш метод `Return` просто возвращает логическое значение `true`.

Вся фактическая работа в `PublishBuilder` выполняется в методе `Bind`. Этот метод принимает два аргумента. Первый – значение,

а второй – функция. В данном примере нас интересует только первый аргумент. Он содержит значение типа размеченного объединения `SendMessageWith`, определяющее желаемую очередь `queueName` и сообщение для отправки в эту очередь. Метод `Bind` использует эти значения для вызова функции `SimpleBus.publish`.

Действие `Create` в контроллере `GuitarsController`, вызываемое для обработки HTTP-запросов типа POST, теперь можно изменить так:

```
[<HttpPost>]
    member this.Create (guitar : Guitar) =
        match base.ModelState.IsValid with
        | false -> guitar |> this.View |> asActionResult
        | true ->
            publish {
                do! SendMessageWith("sample_queue",
                        {Messages.CreateGuitarCommand.Name = guitar.Name})
            }
            base.RedirectToAction("Index") |> asActionResult
```

Концептуально этот пример напоминает вычислительное выражение `trace`, описание которого можно найти в документации MSDN[1], однако обычно вычислительные выражения реализуются с целью упростить решение более сложных задач, чем наша. Дополнительные примеры использования вычислительных выражений можно найти на странице https://github.com/fsharp/fsharpx/blob/master/src/FSharpx.Core/ComputationExpressions/Monad.fs. Дополнительные сведения о создании собственных вычислительных выражений можно найти на странице http://msdn.microsoft.com/ru-ru/library/dd233182.aspx.

В заключение

В этой главе мы рассмотрели множество тем. Мы прошли путь от подготовки первого приложения на основе фреймворка ASP.NET MVC 4 до создания собственного вычислительного выражения, отправляющего сообщения в шину сообщений. Вы также познакомились с некоторыми особенностями языка F#, упрощающими создание веб-приложений., в том числе: размеченные объединения (dis-

[1] http://msdn.microsoft.com/ru-ru/library/vstudio/dd233182%28v=vs.100%29.aspx.

criminated unions), тип `Option`, класс `MailboxProcessor`, сопоставление с образцом (pattern matching), асинхронные потоки операций (async workflows), выражения запросов и многие другие. На протяжении оставшейся части книги вы не раз будете встречать примеры практического применения этих особенностей в других сценариях, поэтому, если что-то осталось за рамками вашего понимания, не волнуйтесь, впереди вас ждет множество примеров.

В следующей главе я познакомлю вас с некоторыми приемами создания веб-служб на языке F#. Основное внимание мы уделим WCF (SOAP), ASP.NET Web API и различным микро-фреймворкам для веб-разработки. Также познакомимся с особенностями модульного тестирования в F#. Как поет Боб Сигер (Bob Seger): «Переверни страницу».

Глава 2. Создание веб-служб на языке F#

Внутри каждой большой программы
есть старающаяся пробиться маленькая программа
– Чарльз Энтони Ричард Хоар (C.A.R. Hoare)

На протяжении моей карьеры, наиболее интригующие меня аспекты, такие как шаблоны проектирования, архитектурные принципы и решения, часто вынуждали меня в первую очередь сконцентрировать свое внимание на уровне служб. Службы, составляющие основу приложения, а иногда и всей системы, казались мне наиболее важным направлением приложения усилий. Со временем мой кругозор значительно расширился, но эти аспекты, которые я считал наиболее интригующими, все еще управляют ходом моих мыслей. Кроме того, службы остаются одним из самых любимых мною направлений в разработке.

В результате перемещения многих шаблонов принципов и приемов, которые мы знаем и любим, на сторону клиента, роль служб значительно изменилась. Несмотря на эти изменения, заставляющие разработчика заниматься разработкой на стороне клиента с тем же усердием, какое прежде он проявлял при создании уровня служб, это не умаляет важность служб, действующих в роли рабочих лошадок. Как бы то ни было, такое перемещение обеспечивает лучшее разделение ответственности и способствует созданию более узкоспециализированных служб, которые лучше подходят для решения поставленных задач. Разбивая решение на мелкие части, прекрасно справляющиеся со своими задачами, мы упрощаем возможность сопровождения, расширения, тестирования и изучения такого решения. Алекс Маккоу (Alex MacCaw) в своей книге «The Little Book on CoffeeScript» (O'Reilly) точно подметил: «Секрет создания крупных и легко сопровождаемых приложений состоит в том, чтобы не создавать крупные приложения».

В этой главе я проведу вас через процесс разработки веб-служб разных типов, написанных главным образом на языке F#. Язык F# обладает множеством особенностей, как нельзя лучше подходящих для создания служб. В этой главе мы также познакомимся с некоторыми фреймворками, которые помогают создавать службы, лучше отвечающие конкретным требованиям. В конце главы я познакомлю вас с несколькими особенностями F#, а так же открытыми библиотеками и фреймворками, упрощающими создание модульных тестов, которые приходится писать в процессе разработки служб.

Установка шаблона проекта WCF

Windows Communication Foundation (WCF) – чрезвычайно гибкое собрание библиотек, обеспечивающих возможность взаимодействий между приложениями и решениями посредством служб. Службы часто создаются с целью стимулировать многократное использование кода, изолировать предметную область решения, взять на себя сложную обработку данных и обеспечить интеграцию с другими системами. Кроме того, службы обычно выполняются в параллельном режиме и зачастую способны извлечь дополнительные выгоды из этого. Если вы ищете инструмент для решения задач, подпадающих под описание выше, обратите внимание на F#.

Как и для большинства примеров в этой книге, лучший способ задействовать F# при создании веб-служб WCF, – создать проект на C# для реализации всего, что связано с точкой входа в службу, и один или более проектов на F#, реализующих внутреннюю логику работы службы. Чтобы получить простой в использовании шаблон проекта, посетите веб-сайт Visual Studio Gallery (http://visualstudiogallery.msdn.microsoft.com/) и найдите шаблон «fsharp wcf». На рис. 2.1 показаны результаты такого поиска. Чтобы установить шаблон, введите имя проекта, который требуется создать, щелкните на кнопке **OK** и затем на копке **Install** (Установить). В результате будет создано решение с тремя проектами:

- Contracts – содержит определения контрактов службы, ее операций и обработки данных.
- Services – содержит реализацию логики службы.
- Web – связывает все вместе и на выходе дает файл(ы) *.svc*.

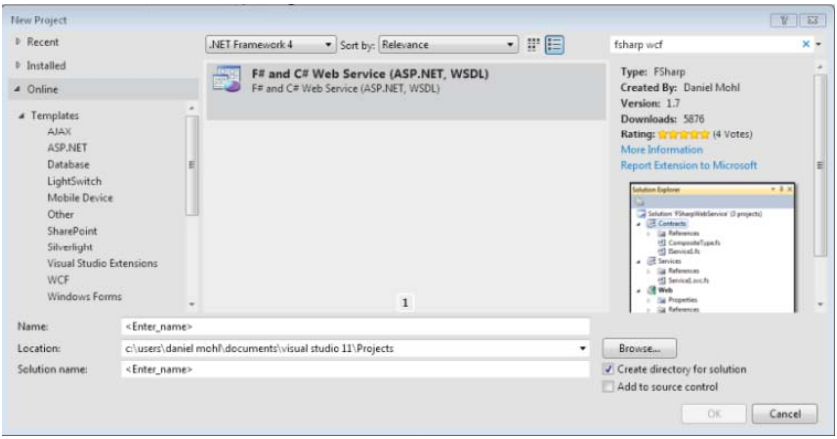

Рис. 2.1. Результат поиска шаблона проекта F# WCF в Visual Studio Gallery

Исследование получившегося решения

Наибольший интерес для нас представляют проекты `Contracts` и `Services`, и далее я буду касаться только их. Несмотря на то, что шаблон разбивает эти две проблемы на отдельные сборки, в действительности это не обязательно. Однако такой подход наиболее часто используется в крупных приложениях. Кроме того, он открывает возможность использовать `ChannelFactory` для создания прямого канала к службе, вместо необходимости применять утилиту *Svcutil.exe*, чтобы сгенерировать код прокси-объекта для клиента. Пример такого подхода я покажу далее в этом разделе.

Проект `Contracts` содержит два файла. Первый включает определение типа `CompositeType`, служащего примером типа записи для передачи исходных данных операциям службы и получения результатов. Как упоминалось в главе 1, всегда следует отдавать предпочтение записям; впрочем, классы F# тоже можно использовать. Содержимое этого файла показано в следующем примере:

```
namespace FSharpWcfServiceApplicationTemplate.Contracts

open System.Runtime.Serialization
open System.ServiceModel

[<DataContract>]
```

```
type CompositeType =
     { [<DataMember>] mutable BoolValue : bool

       [<DataMember>] mutable StringValue : string }
```

> **Примечание.** Шаблон, который мы сейчас рассматриваем, сконструирован так, что может использоваться в обеих версиях, F# 2.0 и F# 3.0. Если вы предполагаете пользоваться только версией F# 3.0, определение типа `CompositeType` можно упростить, использовав атрибут `CLIMutable`. Подробнее об этом атрибуте я расскажу в главе 4.

Как видно в этом примере, к записям можно применять привычные атрибуты WCF, практически так же как к классам. Кроме того, этот пример наглядно показывает, что записи необязательно должны быть неизменяемыми, хотя таковыми они являются по умолчанию. В данном случае определено два изменяемых поля `DataMember`: `BoolValue` и `StringValue`. Хотя в данном случае отсутствие неизменяемости снижает ценность записи, тем не менее, вы все еще сможете пользоваться некоторыми преимуществами записей. Подробнее о преимуществах записей я расскажу чуть ниже.

Другой файл в проекте `Contracts` содержит определение контракта службы `ServiceContract` и связанных с ним контрактов операций `OperationContract`:

```
namespace FSharpWcfServiceApplicationTemplate.Contracts

open System.Runtime.Serialization
open System.ServiceModel

[<ServiceContract>]
type IService1 =
     [<OperationContract>]
     abstract GetData: value:int -> string
     [<OperationContract>]
     abstract GetDataUsingDataContract: composite:CompositeType ->
               CompositeType
```

Этот код определяет интерфейс службы. Важно заметить, что выделенные фрагменты кода не требуются для определения интерфейса на F# в обычных обстоятельствах, но являются обязательными в определениях контрактов WCF `ServiceContract`. Это объясняется тем, что WCF не знает, как обрабатывать безымянные параметры.

Если удалить выделенный код, решение все еще будет компилироваться, но при попытке обратиться к службе будет генерировать сообщение об ошибке «All parameter names used in operations that make up a service contract must not be null. Parameter name: name» (Имена параметров, используемых в операциях, которые составляют контракт службы, не должны быть пустыми. Имя параметра: имя). Параметрам можно давать любые имена по своему усмотрению, но они должны быть обязательно.

Реализация интерфейса `IService1` находится в проекте `Services`. Как обычно, код на F# весьма краток:

```
namespace FSharpWcfServiceApplicationTemplate

open System
open FSharpWcfServiceApplicationTemplate.Contracts

type Service1() =
     interface IService1 with
          member x.GetData value =
               sprintf "%A" value
          member x.GetDataUsingDataContract composite =
               match composite.BoolValue with
               | true -> composite.StringValue <-
                    sprintf "%A%A" composite.StringValue "Suffix"
               | _ -> "do nothing" |> ignore
               composite
```

Вслед за объявлением имени типа `Service1` указывается имя реализуемого интерфейса, и далее следуют реализации всех операций, определяемых интерфейсом. Операция `GetData` просто принимает исходное значение, форматирует его и возвращает результат. В данном случае принимается целое число и возвращается его строковое представление. Этот пример наглядно демонстрирует, что в языке F# имеется богатое, расширяемое семейство типизированных функций, упрощающих форматирование. Подробнее об этом можно узнать на странице: http://msdn.microsoft.com/ru-ru/library/ee370560.aspx.

Операция `GetDataUsingDataContract` принимает запись с именем `composite`. Она сохраняет в поле `StringValue` этого объекта результат форматирования строки – если поле `BoolValue` имеет значение `true` – и возвращает измененную запись вызывающей программе. В целом лучше избегать пользоваться изменяемыми данными, однако ино-

гда это необходимо и данный пример демонстрирует, как этого добиться. Гораздо лучше было бы создать новую запись с желаемыми значениями полей и вернуть ее вызывающей программе. Подробнее об этом будет рассказываться далее в этой главе.

Использование службы

Использовать новую службу можно множеством разных способов. Если служба должна использоваться из приложения на F#, проще всего это реализовать с помощью поставщика типов (type provider) `WsdlService`, входящего в состав F# 3.0. Для этого добавьте ссылку на сборку `FSharp.Data.TypeProviders` и следующий код:

```
open System.ServiceModel
open Microsoft.FSharp.Data.TypeProviders

type webService = WsdlService<"http://localhost:1555/Service1.svc?wsdl">

// Эту строку можно было бы переместить в конфигурационный файл
let serviceUri = "http://localhost:1555/Service1.svc"

let client = new EndpointAddress(serviceUri)
             |> webService.GetBasicHttpBinding_IService1
printfn "The result was %s" <| client.GetData 100
```

> **Примечание.** В зависимости от шаблона проекта, использовавшегося для создания предыдущего примера, вам может также потребоваться добавить ссылку на сборку `System.ServiceModel`.

Поставщик типов `WsdlService` позволяет взаимодействовать с вебслужбой без необходимости использовать отдельный генератор кода, чтобы создать прокси-объект для клиента. В этом примере сначала вызывается поставщик типов `WsdlService`, которому передается URI WSDL-определения службы. Это позволяет ему сгенерировать необходимые промежуточные типы. Далее создается экземпляр `EndPointAddress` и передается методу `GetBasicHttpBinding_IService1`. Это дает возможность использовать один URI в процессе разработки, и переопределять его некоторым внешним значением, например из файла с настройками, при переходе в другое окружение.

Вы могли заметить, что в этом примере поставщик типов `WsdlService` по умолчанию генерирует асинхронные потоки операций. Они

не являются реализацией *модели асинхронного программирования* (Asynchronous Programming Model, APM) (для которой характерно использование парных методов `Begin/End`) которая может быть вам знакома, но они являются операциями, возвращающими экземпляры `Task<'T>`, которые можно использовать в асинхронных потоках операций. Этот прием известен как *шаблон асинхронного программирования на основе заданий* (Task-based Asynchronous Pattern) и, при объединении с асинхронными механизмами F#, существенно упрощает асинхронные обращения к службам. Например:

```
open System.ServiceModel
open Microsoft.FSharp.Data.TypeProviders

type webService = WsdlService<"http://localhost:1555/Service1.svc?wsdl">

// Эту строку можно было бы переместить в конфигурационный файл
let serviceUri = "http://localhost:1555/Service1.svc"

let client = new EndpointAddress(serviceUri)
             |> webService.GetBasicHttpBinding_IService1

seq {1 .. 20}
|> Seq.map (fun i ->
     async {
          let! result = Async.AwaitTask <| client.GetDataAsync i
          do printfn "The result was %s" result
     })
|> Async.Parallel
|> Async.Ignore
|> Async.RunSynchronously
```

Новый код, выделенный жирным шрифтом, начинается с создания последовательности чисел, которая далее передается функции `Seq.map`. Функция `Seq.map` создает новую коллекцию асинхронных вычислений, которые будут вызывать асинхронную версию операции `GetData` и выводить полученные результаты. Поскольку `client.GetDataAsync` возвращает экземпляр `Task<'T>`, мы должны вызвать `Async.AwaitTask`, чтобы преобразовать его в тип `Async<'T>`. Затем асинхронные вычисления настраиваются на параллельное выполнение, их результаты игнорируются, как не представляющие интереса, и затем все это запускается. Вывод из консольного приложения может отличаться от случая к случаю, но, в общем и целом результаты должны выглядеть, как показано на рис. 2.2.

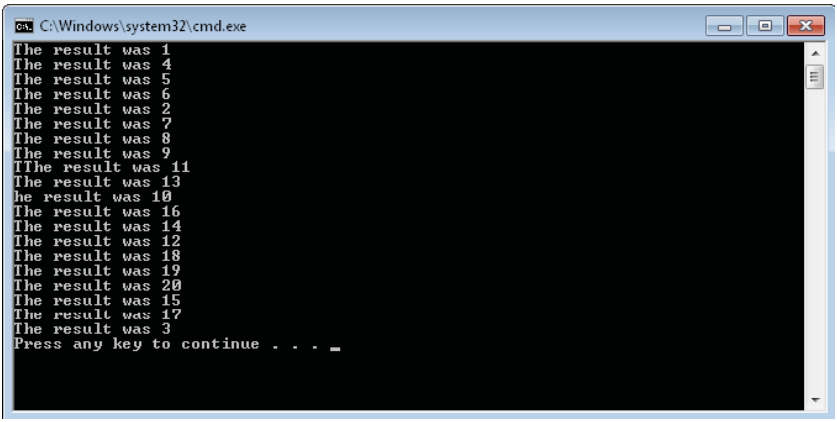

Рис. 2.2. Параллельные вызовы WCF с помощью поставщика типов WsdlService

Кроме приема на основе поставщика типов доступны еще несколько способов использования службы. Один из них основан на использовании утилиты *Svcutil.exe*, генерирующей прокси-объект для клиента. Другой способ опирается на прямое подключение к службе, создаваемое с помощью `ChannelFactory`. Чтобы воспользоваться этим механизмом, сначала нужно добавить ссылки на сборки `Contracts` и `System.Runtime.Serialization`. После этого можно использовать такой код:

```
open System.ServiceModel
open FSharpWcfServiceApplicationTemplate.Contracts

// Эту строку можно было бы переместить в конфигурационный файл
let serviceUri = "http://localhost:1555/Service1.svc"

let address = new EndpointAddress(serviceUri)
let binding = new BasicHttpBinding()
let factory = new ChannelFactory<IService1>(binding, address)
let channel = factory.CreateChannel()
let clientChannel = channel :?> IClientChannel

try
    printfn "The result was %s" <| channel.GetData 100
    clientChannel.Close()
finally
```

```
    if clientChannel.State <> CommunicationState.Closed then
        clientChannel.Abort()
    clientChannel.Dispose()
```

Этот код создает канал, связывающий клиента со службой, после того как будет определен адрес конечной точки и создана привязка. Затем вызывается операция `GetData`, выводится полученный результат и канал закрывается. В заключение проверяется состояние канала. Его работа принудительно прерывается, если он не закрылся, после чего вызывается метод `Dispose`.

Шаблон try/close/catch/abort реализации WCF-клиента

Обычно в языке F# достаточно просто использовать ключевое слово `use`, чтобы обеспечить автоматическое освобождение объектов, реализующих интерфейс `IDisposable`. Однако при работе с WCF существует известная проблема, препятствующая нормальной работе `use`. Это не является чем-то характерным только для F# – эта проблема свойственна именно WCF. По этой причине широкое распространение получил шаблон программирования, известный как try/close/catch/abort. Код на F#, представленный в примере использования фабрики каналов (channel factory), явно помечает занятые им ресурсы, как подлежащие освобождению; однако он недостаточно устойчив к исключениям. Полная реализация данного шаблона могла бы выглядеть, как показано ниже:

```
try
    try
        printfn "The result was %s" <| channel.GetData 100
        clientChannel.Close()
    with
    | :? FaultException as fex -> printfn "Fault: %s" fex.Message
    | ex -> printfn "Error: %s" ex.Message
finally
    if clientChannel.State <> CommunicationState.Closed then
        clientChannel.Abort()
    clientChannel.Dispose()
```

Поскольку часть `with` инструкции является выражением сопоставления с образцом, можно проявить еще более творческий подход и реализовать управление некоторыми аспектами обработки ошибок за пределами самого типа исключения, как показано ниже:

```
try
     // Эту строку можно было бы переместить в конфигурационный файл
     let enableLogging = true

     try
          printfn "The result was %s" <| channel.GetData 100
          clientChannel.Close()
     with
     | :? FaultException as fexl when enableLogging ->
          printfn "Fault Logging: %s" fexl.Message
     | :? FaultException as fex -> printfn "Fault: %s" fex.Message
     | ex -> printfn "Error: %s" ex.Message
finally
     if clientChannel.State <> CommunicationState.Closed then
          clientChannel.Abort()
     clientChannel.Dispose()
```

Шаблон try/close/catch/abort нельзя назвать образцом краткости, поэтому часто его реализацию помещают в отдельный метод/функцию, ради поддержки принципа «не повторяйся» (Don't Repeat Yourself, DRY). На F#, где функции являются сущностями первого рода, сделать это особенно просто. Например:

```
let executeOperation action =
     try
          let enableLogging = true
          try
               action()
               clientChannel.Close()
          with
          | :? FaultException as fexl when enableLogging ->
               printfn "Fault: %s" fexl.Message
          | :? FaultException as fex ->
               printfn "Fault: %s" fex.Message
          | ex -> printfn "Error: %s" ex.Message
     finally
          if clientChannel.State <> CommunicationState.Closed then
               clientChannel.Abort()
          clientChannel.Dispose()

executeOperation (fun () -> printfn "The result was %s"
                                 <| channel.GetData 100)
```

Здесь функция `executeOperation` принимает в аргументе `action` функцию, которую требуется выполнить. Такой подход позволяет написать реализацию шаблона try/close/catch/abort один раз и многократно использовать во всех обращениях к службе.

Погружение в записи

В этой и в предыдущей главах я уже несколько раз упоминал записи. К настоящему моменту те записи, что мы видели в примере реализации службы WCF, были изменяемыми, и я несколько раз повторил, что используя изменяемые записи, вы теряете определенные преимущества. Так в чем же состоят преимущества записей, и какие выгоды несет неизменяемость? Как в примерах, представленных выше, вместо изменяемых использовать неизменяемые данные?

Записи в языке F# – это типы данных, определяющие коллекции именованных значений. Записи можно рассматривать как результат скрещивания классов и структур. Записи, как и классы, являются ссылочными типами, но они поддерживают принцип структурного равенства подобно структурам (которые являются типами значений). Подобно структурам и классам записи могут иметь статические методы и/или методы экземпляров. Одно из существенных отличий записей от классов/структур в том, что записи по умолчанию являются неизменяемыми. Несмотря на то, что при работе с WCF мы вынуждены отказаться от неизменяемых записей, важно знать, какие выгоды теряются при этом. Для начала мы познакомимся с выгодами, которые несет неизменяемость, а затем обсудим некоторые преимущества, которые дают нам записи, даже когда мы вынуждены сделать их изменяемыми.

Итак, что же может дать неизменяемость? Самое важное, пожалуй, преимущество неизменяемости в том, что она упрощает реализацию параллельных алгоритмов. На F# параллельные приложения создаются легко и просто, но проблемы начинают возникать, как только появляются изменяемые объекты, совместно используемые и модифицируемые несколькими потоками выполнения. При использовании неизменяемых объектов эта проблема автоматически снимается.

Неизменяемость также обеспечивает надежность тестирования и гарантирует, что тесты всегда будут возвращать предсказуемые результаты. Она упрощает изучение кода и уменьшает вероятность появления сложных в обнаружении ошибок, вызванных неожиданным изменением данных. Еще одно преимущество состоит в том, что неизменяемость дает компилятору свободу переупорядочения кода во время компиляции, чтобы повысить его эффективность.

Да, неизменяемые записи предпочтительнее, но их использование в нашей службе WCF приведет к появлению исключения. Но, даже при том, что в данном примере мы теряем выгоды, которые дает не-

изменяемость, применение записей как таковых все же дает определенные преимущества. В их число входит поддержка сопоставления с образцом, а также выразительность синтаксиса определения, создания и копирования. Воспользуемся этими особенностями, чтобы получить дополнительные выгоды от использования изменяемой записи в нашей службе WCF. Ниже представлена измененная версия метода `GetDataUsingDataContract`:

```
member x.GetDataUsingDataContract composite =
    match composite with
    | { BoolValue = true; StringValue = _ } ->
        { composite with StringValue = "My New String" }
    | _ -> composite
```

Теперь этот метод выполняет сопоставление записи `composite` с образцом. Если поле `BoolValue` исследуемой записи имеет значение `true`, то независимо от значения поля `StringValue` создается новая запись типа `CompositeType` путем копирования исходного объекта `composite` и присваиванию полю `StringValue` значения `"My New String"`. Так как в выражении сопоставления с образцом фактически проверяется только условие `BoolValue = true`, его можно упростить:

```
member x.GetDataUsingDataContract composite =
    match composite with
    | { BoolValue = true } ->
        { composite with StringValue = "My New String" }
    | _ -> composite
```

Создание службы ASP.NET Web API

Фреймворк ASP.NET MVC 4 включает новую особенность, которая называется ASP.NET Web API и используется для создания современных HTTP-служб. ASP.NET Web API – это дальнейшее развитие различных REST API, разрабатывавшихся командой WCF в последние несколько лет и объединенных со многими замечательными особенностями фреймворка ASP.NET MVC. В результате такого слияния получилась превосходная платформа для создания HTTP-служб, поддерживающих такие особенности, как возможность согласования типа содержимого с клиентом (например, XML, JSON, и др.), маршрутизация, соглашения, простота тестирования и многие другие.

В главе 1 вы могли заметить, что мастер создания проекта веб-приложения на основе ASP.NET MVC 4 предлагает на выбор еще один шаблон. Создание HTTP-службы на основе ASP.NET Web API и F# не вызывает никаких сложностей, благодаря все тому же расширению для Visual Studio. Если в ходе обсуждения ASP.NET MVC вы установили расширение для Visual Studio, тогда, чтобы создать решение F#/C# ASP.NET Web API, достаточно открыть диалог мастера создания проекта, выбрать **Visual F# → ASPNET → F# and C# Web Application (ASP.NET MVC 4)**. Затем выбрать **WebApi Project** и щелкнуть на кнопке **OK**.

> **Примечание.** К моменту написания этих строк, шаблон проектов F#/C# ASP.NET Web API поддерживал только механизм представлений Razor. Поле выбора механизма представлений запрещено и будет показывать выбор, сделанный ранее. Независимо от того, что отображается в этом поле, будет использоваться механизм представлений Razor.

На рис. 2.3 изображен диалог выбора типа проекта непосредственно перед щелчком на кнопке **OK**.

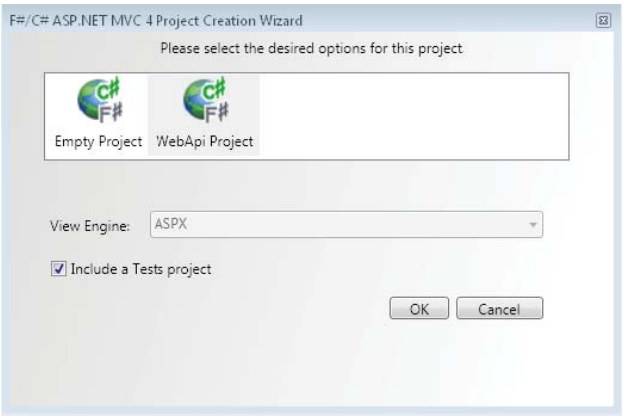

Рис. 2.3. Мастер создания проекта F#/C# ASP.NET Web API

Анализ шаблона

Шаблон проекта F#/C# ASP.NET Web API делится на две основные части:

1. ASP.NET MVC – этот шаблон включает типичное приложение ASP.NET MVC с готовым представлением Home и контроллером `HomeController`. В примере приложения модель данных не нужна, поэтому она отсутствует в шаблоне. Однако ее легко можно добавить обычным способом. Часть ASP.NET MVC проекта является необязательной. Ее можно полностью удалить, если вы собираетесь заняться разработкой HTTP-служб(ы).
2. ASP.NET Web API – наряду с типичным приложением ASP.NET MVC содержит HTTP-службу, основанную на ASP.NET Web API.

Я уже демонстрировал, как создать приложение ASP.NET MVC с использованием F#, поэтому я не буду тратить время на повторный обзор этой части шаблона. Часть ASP.NET Web API шаблона содержит пример готовой службы ASP.NET Web API. При беглом осмотре код HTTP-службы легко можно спутать с контроллером для работы с фреймворком ASP.NET MVC. Однако при более детальном изучении выясняются следующие отличия:

- отсутствует экземпляр `View` , соответствующий контроллеру;
- отличается содержимое файла *Global.fs*;
- контроллер наследует другой базовый класс, а имена его методов соответствуют методам протокола HTTP.

Как и в других шаблонах, рассматривавшихся выше, серверная часть этого шаблона написана на языке F#.

С точки зрения преимуществ, единственное, что пока дает использование языка F# в данном простом шаблоне – это краткость. Но как только вы приступите к реализации более сложных функциональных возможностей, теперь в вашем распоряжении будет вся мощь языка F#.

Различия в файле *Global.fs* невелики. Во-первых, определен новый тип записи с именем `MapHttpRouteSettings`, который будет использоваться ниже. Объявление типа выглядит, как показано ниже:

```
type MapHttpRouteSettings = { id : obj }
```

В классе `Global` появился один новый статический метод с именем `RegisterGlobalFilters`, добавляющий экземпляр `HandleErrorAttribute` в глобальную коллекцию фильтров:

```
static member RegisterGlobalFilters(filters:GlobalFilterCollection) =
    filters.Add(new HandleErrorAttribute())
```

Нам необходимо предоставить информацию о маршрутизации запросов в HTTP-службе, поэтому в метод `RegisterRoutes` добавлен вызов метода с именем `MapHttpRoute`. В этой карте маршрутов не требуется определять все маршруты, потому что по соглашению операции ASP.NET Web API отображаются в HTTP-методы. Данный вызов выглядит так:

```
routes.MapHttpRoute("DefaultApi", "api/{controller}/{id}",
    {id = RouteParameter.Optional}) |> ignore
```

Наконец, во все методы, вызываемые методом `Start`, добавлены вызовы статического метода `RegisterGlobalFilters`.

Основное отличие контроллера в том, что он наследует класс `ApiController`, вместо `Controller` и, как упоминалось выше, имена методов соответствуют именам методов протокола HTTP. Ниже представлено определение `ValuesController`:

```
namespace FsWeb.Controllers

open System.Web
open System.Web.Mvc
open System.Net.Http
open System.Web.Http

type ValuesController() =
    inherit ApiController()

    // GET /api/values
    member x.Get() = [| "value1"; "value2" |] |> Array.toSeq
    // GET /api/values/5
    member x.Get (id:int) = "value"
    // POST /api/values
    member x.Post ([<FromBody>] value:string) = ()
    // PUT /api/values/5
    member x.Put (id:int) ([<FromBody>] value:string) = ()
    // DELETE /api/values/5
    member x.Delete (id:int) = ()
```

Если запустить проект, в браузере отобразится страница начального представления. Перемещаясь по URL, таким как http://localhost:**[port]**/api/values, можно узнать, что получается в результате вызова метода `Get`.

Если у вас появится желание создать автономную службу (self-host service), вы так же легко сможете сделать это. Например, чтобы быстро создать консольное приложение, реализующее службу, представленную выше, нужно сначала добавить ссылки на сборки `System.Net`, `System.Net.Http`, `System.Web.Http` и `System.Web.Http.SelfHost`. После этого можно использовать код, как показано ниже:

```
namespace WebApiServiceHost

open System
open System.Web.Http
open System.Web.Http.SelfHost

type ValuesController() =
     inherit ApiController()
     // GET /api/values
     member x.Get() = [| "value1"; "value2" |] |> Array.toSeq

module ServiceHost =
     type MapHttpRouteSettings = { id : obj }

     let main args =
          use config =
               new HttpSelfHostConfiguration(Uri "http://localhost:8080/")
          config.Routes.MapHttpRoute("DefaultApi", "api/{controller}/{id}",
               {id = RouteParameter.Optional}) |> ignore

          use server = new HttpSelfHostServer(config)
          server.OpenAsync().Wait()

          printfn "%s\r\n%s" "I'm ready to serve..."
               "Go to http://localhost:8080/api/values in a browser"
          Console.ReadLine() |> ignore

     main ()
```

Примечание. Контроллеры должны объявляться в пространстве имен, а не в модуле.

Самое замечательное в возможности использовать F# для создания таких служб состоит в том, что теперь можно использовать все замечательные особенности языка и примеры, представленные выше (и которые еще будут показаны ниже), для создания превосходных

HTTP-служб, возвращающих данные в JSON или каком-нибудь другом формате. Добавьте мощь языка F# к стандартным способам увеличения масштабируемости, таким как кеширование, распределенные вычисления и распределение нагрузки, и вы получите победоносную комбинацию, позволяющую службам легко разрастаться в масштабах. Рассмотрим несколько способов использования таких служб.

Взаимодействие с HTTP-службой

Одним из типичных способов взаимодействий с HTTP-службами является непосредственный вызов из сценариев на JavaScript, выполняющихся в браузере. Поскольку этот прием несколько выходит за рамки данной книги, я не буду подробно останавливаться на нем. Однако, чтобы увидеть, насколько это просто, взгляните на следующий пример:

```
<!DOCTYPE html>
<html lang="en">
<head>
     <meta charset="utf-8" />
     <title>ASP.NET Web API</title>
     <link href="@Url.Content("~/Content/Site.css")"
          rel="stylesheet" type="text/css" />
     <meta name="viewport" content="width=device-width" />
</head>
<body>
     <div id="sampleValues"/>

     <script src="@Url.Content("~/Scripts/jquery-1.6.2.min.js")"
          type="text/javascript"></script>
     <script
          src="@Url.Content("~/Scripts/modernizr-2.0.6-development-only.js")"
          type="text/javascript"></script>
     <script type="text/javascript">

          $(function() {
               $.getJSON( '/api/values' ).then(function( values ) {
                    var sampleValues = [];

                    $.each( values, function( i, value ) {
                         sampleValues.push( '<div>' + value + '</div>' );
                    });
```

```
                $( '#sampleValues' ).html( sampleValues.join('') );
            });
        });

    </script>
</body>
</html>
```

Примечание. Этот пример был создан путем модификации файла Index.cshtml, входящего в состав шаблона F#/C# ASP.NET Web API.

Для вызова метода `Get` контроллера `ValuesController` в этом примере используется функция `getJSON` из библиотеки jQuery. Результаты, полученные в ответ, просто выводятся на экран. Совершенно очевидно, что в этом примере не хватает некоторых тонкостей, связанных с обработкой и форматированием получаемых результатов, тем не менее, он достаточно наглядно демонстрирует, насколько просто взаимодействовать с нашей новой службой из JavaScript.

Но, что делать, если потребуется обратиться к службе со стороны сервера? Как обычно, F# дает на выбор несколько возможностей! Рассмотрим их по порядку...

С использованием объекта HttpClient

Возможность взаимодействия со службой из JavaScript безусловно важна, однако иногда бывает необходимо обратиться к службе со стороны сервера. Проще всего для этой цели использовать `System.Net.WebRequest`, `System.Net.WebClient` или `System.Net.Http.HttpClient`. Каждый из этих вариантов имеет свои достоинства и недостатки, но, на мой взгляд, `HttpClient` позволяет получить больше возможностей с меньшим объемом кода. Чтобы попробовать обратиться к нашей службе с помощью `HttpClient`, создайте новое консольное приложение на языке F# и установите NuGet-пакет `Microsoft.AspNet.WebApi.Client` или добавьте ссылки на сборки `System.Net.Http`, `System.Net.Http.Formatting` и библиотеку `Json.NET`.

Примечание. Библиотека `System.Json`, как известно, работает довольно медленно. В бета-версиях фреймворка ASP.NET MVC 4 по умолчанию использовалась именно `System.Json`. Однако в предвыпускной версии (release candidate) ее заменили библиотекой `Json.NET`. Дэйв Уорд (Dave Ward) подробно рассказывает об этом по адресу: http://encosia.com/jquery-asp-net-web-api-and-json-net-walk-into-a-bar/.

Теперь можно добавить очень простую реализацию, как показано ниже:

```
open System
open System.Net.Http
open System.Json

let client = new HttpClient()

async {
     let! res = Async.AwaitTask
                  <| client.GetAsync("http://localhost:5551/api/values")
     let! content = Async.AwaitTask <| res.Content.ReadAsStringAsync()
     printfn "The result is %s" content
} |> Async.Start

printfn "Please Wait..."
Console.ReadKey() |> ignore
```

Для обращения к службе и ожидания результата этот пример использует метод `GetAsync`. Затем он асинхронно читает содержимое ответа. Запустив этот пример можно быстро убедиться в его асинхронной природе, так как сразу после запуска выводится строка «Please Wait...», а результат обращения к службе – немного погодя.

Предыдущий пример работает, но взгляните, что произойдет, если перегруженный метод `Get` в классе `ValuesController` будет выглядеть, как показано ниже, – он возвращает строку, если на входе получает значение 1 или 2 в аргументе `id`, и генерирует ошибку 404 для любых других значений:

```
member x.Get (id:int) =
     match id with
     | 1 | 2 -> sprintf "Value is %i" id
     | _ -> raise <|
          HttpResponseException(
               new HttpResponseMessage(HttpStatusCode.NotFound))
```

После такого изменения службы пример клиента, представленный выше, остановится в ожидании строки, если, например, URI будет иметь такой вид: http://localhost:5551/api/values/3. Существует несколько вариантов решения этой проблемы. Один из них – до-

бавить вызов метода EnsureSuccessStatusCode ответа. Затем мы сможем просто заключить вызов в инструкцию try/with, как показано ниже:

```
async {
    try
        let! res = Async.AwaitTask
                   <| client.GetAsync("http://localhost:5551/api/values/3")
        res.EnsureSuccessStatusCode() |> ignore
        let! content = Async.AwaitTask <| res.Content.ReadAsStringAsync()
        printfn "The result is %s" content
    with
    | ex -> printfn "%s" ex.Message
} |> Async.Start
```

Другой способ – реализовать обработку ошибок в стиле продолжений. Например:

```
let task = async {
    let! res = Async.AwaitTask
               <| client.GetAsync("http://localhost:1232/api/values/3")
    res.EnsureSuccessStatusCode() |> ignore
    return! Async.AwaitTask <| res.Content.ReadAsStringAsync()
}

Async.StartWithContinuations(
    task,
    (fun r ->
        seq { yield "The returned values are:" }
        |> Seq.append <| JsonConvert.DeserializeObject<seq<string>>(r)
        |> Seq.reduce (fun acc v -> sprintf "%s %s" acc v)
        |> printfn "%s"),
    (fun err ->
        printfn "The request was NOT successful with error %s" err.Message),
    (fun canc -> printfn "The request was cancelled"))
```

Метод Async.StartWithContinuations принимает асинхронное задание в первом аргументе. Во втором, третьем и четвертом аргументах передаются функции, определяющие операции, которые должны быть выполнены по завершении асинхронного задания, в зависимости от ее результата. Если задание завершилось успешно, вызывается первая из трех функций. Если возникла ошибка, вызывается вторая функция. А в случае отмены задания – третья.

Код функции, вызываемой в случае успеха, стоит обсудить подробнее, так как он являет собой пример использования библиотеки `Json.NET` для десериализации ответа, а также пример объединения строк без использования строковых переменных. Первая строка кода в функции создает последовательность с начальным значением строки. Затем выполняется десериализация ответа в последовательность строк и добавление ее в конец начальной последовательности, в результате чего создается новая последовательность, содержащая объединяемые значения. Далее выполняется цикл по последовательности строк и производится ее свертка до единственной строки, путем добавления каждого нового значения в аккумулятор. Наконец накопленный результат вводится на экран.

Поставщик типов JSON

Подход на основе `HttpClient` действует очень хорошо, но он может доставлять некоторые неудобства при работе с данными в формате JSON или XML, возвращаемыми службой. Чтобы обеспечить надежную обработку таких результатов, необходимо преобразовать результат в объект или коллекцию объектов. Если вы контролируете обе стороны, это не станет большой проблемой, потому что у вас уже имеются записи или классы модели данных.

Гораздо сложнее, когда реализация службы недоступна. Представьте, например, что вам требуется создать клиента для взаимодействия со службой iTunes. Так как iTunes API возвращает данные в формате JSON, вам придется вручную создавать записи или классы, вызывать HTTP-службу iTunes и отображать результаты этого вызова в записи или классы. Было бы намного проще, если бы имелся способ пользоваться преимуществами строгой типизации без необходимости вручную определять инфраструктуру модели данных.

Как упоминалось выше, в версии F# 3.0 поддерживается новая особенность с названием поставщики типов (type providers), которая является именно тем, что нам нужно. Поставщики типов, обсуждавшиеся выше, являются стандартной частью языка, а я хочу рассказать о поставщике типов, созданном в сообществе. Он доступен в составе открытой коллекции библиотек, известной как FSharpx (https://github.com/fsharp/fsharpx). Конкретная библиотека, содержащая провайдеры типов для JSON, называется `FSharpx.TypeProviders` и доступна в NuGet под тем же названием. Получить дополнительную информацию об этом поставщике типов можно

по адресу: http://www.navision-blog.de/2012/03/25/typed-access-to-json-and-xml/.

Применение поставщика типов для работы с данными в формате JSON, чтобы обеспечить строго типизированный доступ к результатам простого поиска в службе iTunes, осуществляется очень просто. Сначала необходимо обратиться к URI службы iTunes, чтобы получить образец возвращаемых результатов. Эти результаты можно затем встроить в определение типа или поместить их в файл, на который будет ссылаться определение типа. После этого достаточно просто вызвать службу и обратиться к результатам. Ниже показан пример использования версии `FSharpx.TypeProviders` выше 1.5.8:

```
open System
open System.Net.Http
open FSharpx

type itunesApi = StructuredJSON<FileName="itunesSample.json">

let client = new HttpClient()

async {
     let! res = Async.AwaitTask
                  <| client.GetAsync(
                        "http://itunes.apple.com/search?term=guitar&limit=5")
     let! content = Async.AwaitTask <| res.Content.ReadAsStringAsync()
     let root = itunesApi(documentContent = content).Root
     printfn "%i results were found." root.ResultCount

     root.GetResults()
     |> Seq.iter(fun x ->
          printfn "Artist Name is '%s' and Collection Name is '%s'"
               x.ArtistName x.CollectionName)
} |> Async.Start

printfn "Please Wait..."
Console.ReadKey() |> ignore
```

Прежде чем покинуть ASP.NET Web API

Знакомясь с различными клиентами, которые мы создавали, вы, возможно, заметили, как их реализации можно уложить в функциональную парадигму программирования. Единственное, что пока выражено недостаточно очевидно, – большинство аспектов фрейм-

ворка ASP.NET Web API, включая серверные программные компоненты, реализованы в функциональном стиле. Райан Райли (Ryan Riley) написал отличную статью, раскрывающую некоторые подробности. Как он отмечает, при работе с ASP.NET Web API в основе всего лежит `HttpMessageHandlers`. Для построения цепочек обработчиков этот класс использует шаблон делегирования. Данный шаблон находится в одном ряду со многими понятиями, которые я уже упоминал в этой книге.

Мы еще вернемся к ASP.NET Web API далее в этой главе, когда будем рассматривать тему тестирования.

Другие веб-фреймворки

ASP.NET Web API – отличный фреймворк для создания HTTP-служб, но кому-то он может не подходить по тем или иным причинам. В этом разделе я перечислю некоторыми другими популярными альтернативами, позволяющими создавать HTTP-службы на F#.

Service Stack

Service Stack – открытый фреймворк для веб-служб, который можно найти по адресам: https://github.com/ServiceStack/ServiceStack и http://www.servicestack.net/. Его основное предназначение – служить основой для высокопроизводительных веб-служб, управляемых шаблонами (pattern-driven), создаваемых по принципу «сначала код» (code-first). Под зонтиком Service Stack приютилось также несколько вспомогательных проектов, тесно связанных с ядром Service Stack, в числе которых можно назвать `ServiceStack.Text`, `ServiceStack.Redis` и `Service Stack.OrmLite`.

Реализация минимальной веб-службы на языке F# и на основе Service Stack, выглядит очень просто. Для этого нужно лишь установить NuGet-пакет `ServiceStack` в новый проект приложения на F# и добавить ссылку на сборку `System.Web`. Это позволит реализовать автономную службу, как показано ниже:

```
open System
open ServiceStack.ServiceHost
open ServiceStack.WebHost.Endpoints
open ServiceStack.ServiceInterface

[<RestService("/api/values/{Id}")>]
```

```
type ValueRequest = { mutable Id : int }

type ValueResponse = { Result : string }

type ValuesService() =
     inherit RestServiceBase<ValueRequest>()
     override x.OnGet request =
          { Result = sprintf "The value is %i" request.Id } |> box

type AppHost() =
     inherit AppHostHttpListenerBase("ValuesService",
                                     typeof<ValuesService>.Assembly)
     override this.Configure container = ()

module ServiceHost =
     let main args =
          use appHost = new AppHost()
          appHost.Init()
          appHost.Start "http://localhost:9090/"
          printfn "The service has started on port 9090"
          Console.ReadLine() |> ignore

     main ()
```

Первое, на что следует обратить внимание в этом фрагменте – определение типа `ValueRequest` *объекта передачи данных* (Data Transfer Object, DTO). Маршрут запроса в этом примере устанавливается как атрибут этого объекта. Как можно предположить, исходя из имени, объект `ValueRequest` представляет входные данные для обращения к данной службе. Каждое именованное значение в записи F# должно быть помечено как изменяемое (`mutable`), чтобы ее можно было использовать с инфраструктурой Service Stack. При желании вместо записи можно использовать класс. Кроме того, имена значений должны соответствовать именам любых замещаемых параметров в маршруте.

Тип `ValuesService` представляет главную реализацию HTTP-службы. В данном случае мы наследуем класс `RestServiceBase`, чтобы обеспечить поддержку функциональности службы, но с тем же успехом можно было бы унаследовать класс `ServiceBase` или реализовать интерфейс `IService<T>`. Переопределенная версия метода `OnGet` будет вызываться при получении `GET`-запроса. В данном случае метод просто создает экземпляр записи `ValueResponse` и возвращает его.

Тип AppHost реализует настройку службы. Наконец, код в модуле ServiceHost создает службу, определяет порт для приема запросов и запускает весь механизм в работу.

Если вы не сторонник применения атрибутов для декорирования объектов запросов, можете определить маршруты, как это делалось в примере службы на основе фреймворка ASP.NET Web API. Для этого измените тип AppHost, как показано ниже, и удалите атрибут из объявления типа ValueRequest:

```
type AppHost() =
     inherit AppHostHttpListenerBase("ValuesService",
                                      typeof<ValuesService>.Assembly)
     override this.Configure container =
          base.Routes.Add<ValueRequest>("/api/values/{Id}") |> ignore
```

Чтобы получить возможность использовать службу, установите NuGet-пакет ServiceStack и добавьте код инициализации, как показано ниже:

```
open System
open ServiceStack.ServiceClient.Web

type ValueResponse = { mutable Result : string }

let client = new JsonServiceClient("http://localhost:9090")

client.GetAsync<ValueResponse>("/api/values/1",
     (fun r -> printfn "Result is %s" r.Result),
     (fun r ex -> raise ex) )

printfn "Please Wait..."
Console.Read() |> ignore
```

К этому примеру практически нечего добавить. Здесь мы создаем экземпляр клиента типа JsonServiceClient, но существует множество других похожих типов, включая: XmlServiceClient, WcfServiceClient и Soap12ServiceClient. Полный их перечень вместе с исходными текстами можно найти на сайте GitHub: https://github.com/ServiceStack/ServiceStack/tree/master/src/ServiceStack.Common/ServiceClient.Web.

Создав клиента, мы можем обратиться к службе несколькими способами, в зависимости от типа клиента. В данном случае мы

используем асинхронные версии методов с именами, соответствующими стандартным методам протокола HTTP. Здесь мы еще раз видим пример реализации в стиле продолжений. Первый аргумент определяет маршрут вызова. Второй определяет функцию, которая будет выполнена в случае успешного завершения запроса. Ей будет передан ответ службы в виде аргумента, который в данном примере будет иметь тип `ValueResponse`. Последний аргумент определяет функцию для вызова в случае ошибки. Ей будут переданы ответ и исключение. В данном примере мы просто повторно возбуждаем полученное исключение.

> **Примечание.** Когда реализации клиента и службы находятся в руках одного разработчика, общие типы принято помещать в отдельную сборку, совместно используемую обоими проектами, чтобы не дублировать их определения, как это сделал я с типом `ValueResponse`. Служба не требует, чтобы тип `ValueResponse` был изменяемым, но это необходимо клиенту. Поэтому в общей сборке тип должен быть определен как изменяемый.

Nancy

Nancy – открытый фреймворк для создания HTTP-служб на платформе .NET. Он разрабатывался, что называется, «с нуля» и предоставляет простой и легковесный механизм управления службами, который может работать практически где угодно. Кроме того, он поддерживает возможность интеграции с большим количеством механизмов представлений, что позволяет использовать Nancy как для создания автономных HTTP-служб, так и в роли полноценного веб-фреймворка.

Чтобы запустить простую автономную службу, создайте новое консольное приложение на F#, установите NuGet-пакет `Nancy.Hosting.Self` и добавьте следующий код:

```
open System
open Nancy
open Nancy.Hosting.Self

let GetNancyParam prms key = (prms :> DynamicDictionary).[key]

type System.String with
     static member toNancyResponse (s:string) = Response.op_Implicit s

type ValuesModule() =
```

```
    inherit NancyModule()
    let getResponse v = v |> sprintf "The response is %A"
                          |> String.toNancyResponse
    do base.Get["/api/values"] <-
         fun _ -> ["value1", "value2"] |> getResponse
    do base.Get["/api/values/{id}"] <-
         fun prms -> (unbox prms, "id") ||> GetNancyParam |> getResponse

let host = NancyHost( Uri "http://localhost:9191" )
host.Start()

printfn "The service is running on port 9191..."
Console.ReadKey() |> ignore

host.Stop()
```

С точки зрения количества строк, это одна из самых коротких реализаций, виденных нами до сих пор. Однако она же является самой сложной. С другой стороны, она достаточно легко читается, особенно если вы имеете полное представление о происходящем здесь. Давайте разбираться...

Первая функция, с именем `GetNancyParam`, является вспомогательной и используется для получения значения параметра из экземпляра типа `DynamicDictionary`, который используется фреймворком Nancy для хранения значений, поступивших вместе с запросом в виде параметров в маршрутах. В языке F# не принято автоматически изменять какие-либо аспекты (такие как признак изменяемости). Вместо этого от разработчика требуется явно указать, что и как нужно изменить. Примером тому является требование явно выполнить приведение к требуемому типу или интерфейсу. Функция `GetNancyParam` явно приводит аргумент `prms` (сокращенно от «parameters») к типу `DynamicDictionary`, а затем извлекает значение указанного ключа `key` (в данном случае `id`).

Следующие несколько строк демонстрируют еще одну особенность языка F#, о которой я пока не рассказывал, – *расширение типов* (type extensions). Механизм расширения типов дает возможность добавлять новые функциональные возможности в существующие типы. В данном примере реализуется расширение типа `System.String` возможностью преобразовывать строки в объект `Response` фреймворка Nancy. Это преобразование необходимо по той простой причине, что фреймворк Nancy позволяет возвращать только объекты `Response`

или один из нескольких операторов неявного преобразования типа. Дополнительную информацию об этом можно найти на странице: https://github.com/NancyFx/Nancy/wiki/Defining%20routes.

Далее следует определение класса модуля Nancy. Модули Nancy – это основной инструмент создания служб. Как видно в примере, чтобы создать модуль Nancy, достаточно определить класс, наследующий класс `NancyModule`. Первая функция внутри класса помогает преобразовывать исходное значение в строку для последующего вывода. Она демонстрирует еще одну замечательную особенность F#. Область видимости данной функции ограничивается рамками класса. Такие функции известны также как *замыкания* (closures). Замыкания часто используются для инкапсуляции функциональности и/или состояния. Кроме того, они могут быть очень полезны тем, кто стремится следовать принципу «не повторяйся» (Don't Repeat Yourself, DRY).

Ниже следуют определения маршрутов. При работе с фреймворком Nancy маршруты определяются в конструкторах классов, поэтому для их настройки в F# следует использовать *привязку do* (do binding). Привязки `do` могут быть статическими или нестатическими и выполняются в порядке следования внутри определения класса. Первая привязка определяет маршрут, возвращающий строковое представление списка строк.

> **Примечание.** В языке F# поддерживается возможность снабжать определения конструкторов дополнительным ключевым словом `with`, инициализирующим различные аспекты объекта. Однако привязка `do` является более предпочтительным способом.

Второй маршрут возвращает строку, включающую значение `id`, полученное в запросе. Здесь есть пара интересных аспектов. Во-первых, для приведения экземпляра ссылочного типа обратно к типу значения используется функция `unbox`. Полученное таким способом значение затем передается функции `GetNancyParam`, которая в свою очередь приводит его к типу `DynamicDictionary`. Второй интересный момент – использование *прямого конвейерного оператора с двумя аргументами* (`||>`). Он принимает кортеж с двумя значениями слева, извлекает эти значения и передает функции справа в виде двух отдельных аргументов. Эту строку можно было бы записать иначе:

```
fun prms -> GetNancyParam (unbox prms) "id" |> getResponse
```

Оба фреймворка, Nancy и Service Stack, прекрасно уживаются с языком F#, благодаря чему вы можете использовать F# для создания надежных HTTP-служб на их основе. Фактически, выразительность и удобочитаемость кода на F#, делает этот язык отличным выбором, даже если нет других причин выбрать именно его. С другой стороны, эти два фреймворка следуют объектно-ориентированной парадигме, что не позволяет использовать наиболее сильные стороны F#. Однако для желающих создавать веб-стеки в более идиоматичном для F# стиле существуют альтернативные решения. В их число входят: реализация предметно-ориентированного языка для веб-разработки Figment (https://github.com/mausch/Figment) Маурисио Шеффера (Mauricio Scheffer), библиотека PicoMvc (https://github.com/robertpi/PicoMvc) Роберта Пикеринга (Robert Pickering) и библиотека комбинаторов Frank (https://github.com/frank-fs/frank) Райана Райли (Ryan Riley). Из-за необходимости экономии места в книге, я приведу пример использования только одного из них.

Примечание. Важно отметить, что от вас не требуется быть мастером функционального программирования, чтобы использовать F#. Как отмечалось в предыдущем абзаце, вы можете выбрать этот язык, только чтобы пользоваться его выразительностью и наличием механизма автоматического определения типов. Нет ничего неправильного в том, что вы выберете и будете использовать только интересующие вас особенности языка. Как только вы начнете пользоваться языком F#, вы часто будете обнаруживать, что он подталкивает вас на «правильный» путь. Если вам интересно узнать, как сделать свой код более функциональным, обращайтесь к статье Ричарда Минерича (Richard Minerich) по адресу: http://bit.ly/functional-programming.

Frank

Первоначально проект Frank задумывался как фреймворк, такой как Sinatra, но основанный на принципах и приемах функционального программирования. Он была объединен с библиотекой под названием Frack, реализующей промежуточную (middleware) функциональность, дополняющую веб-фреймворки, такие как Frank. Однако Райан Райли (Ryan Riley), создатель проектов Frank и Frack, а также научный редактор этой книги, поняв, что фреймворк ASP.NET Web API действительно движется в правильном направлении, превратил проект Frank в простую функциональную обертку вокруг библиотеки `System.Net.Http`, основанной на приеме композиции функций. В настоящее время проект Frack остановился в развитии, но многие его особенности были перенесены в проект Frank.

Ниже приводится пример простой автономной службы на основе Frank:

```
open System
open System.Net
open System.Net.Http
open System.Web.Http
open System.Web.Http.SelfHost
open Frank

let values request =
     respond HttpStatusCode.OK
          <| new StringContent(["values1"; "values2"].ToString())
          <| ignore
     |> async.Return

let value request =
     let id = getParam request "id"
     respond HttpStatusCode.OK
          <| new StringContent(sprintf "The value is %s" id)
          <| ignore
     |> async.Return

let app = merge [ route "/api/values" <| get values
                  route "/api/values/{id}" <| get value ]

module HostServer =
     let main args =
          let baseUri = "http://localhost:9393"
          use config = new HttpSelfHostConfiguration(baseUri)
          config.Register app

          use server = new HttpSelfHostServer(config)
          server.OpenAsync().Wait()

          printfn "The service is running at %s..." baseUri
          Console.ReadKey() |> ignore
          server.CloseAsync().Wait()

main ()
```

В этом примере выделены наиболее важные строки. Рассмотрим их подробнее, функцию за функцией. Первая функция, с именем `values`, определяет, что будет возвращаться при обращении к маршруту `/api/values`.

Функция values делится на две основные части. Первая определяет ответ, который должен быть отправлен на конкретный запрос. Этот ответ конструируется с помощью обратного конвейерного оператора (backward pipe operator), образующего композицию функций, и функции respond из библиотеки Frank. Функции respond передается код состояния HTTP 200 (OK) и строковое представление массива строк. Последнее, что передается функции respond – это функция, которая добавляет различные заголовки в ответ. Библиотека Frank имеет множество таких встроенных функций; однако в данном случае мы не нуждаемся в них, поэтому передаем функцию ignore.

Вторая часть функции values – конвейер для передачи ответа функции async.Return. Эта функция является частью типа AsyncBuilder в F#. Для достижения того же эффекта первую часть можно было бы просто завернуть в асинхронный блок, как показано в следующем примере:

```
let values request = async {
    return respond HttpStatusCode.OK
        <| new StringContent(["values1"; "values2"].ToString())
        <| ignore
}
```

Функция value имеет реализацию, очень похожую на реализацию функции values. Но в отличие от последней она может реагировать соответственно на значения параметров в URI. Функция getParam входит в состав библиотеки Frank и обеспечивает все необходимое ля этого. Достаточно передать ей имя параметра в маршруте (в данном случае id) и она вернет значение этого параметра в URI. Служба в данном примере просто возвращает клиенту значение, которое было указано в URI, но в общем случае логика может быть сколь угодно сложной.

Последний интересный момент – объявление маршрутов. В рассматриваемом примере переменной app присваивается результат вызова функции merge, входящей в состав библиотеки Frank. Она принимает последовательность значений типа HttpResource. Значения этого типа включают информацию о маршруте, перечень HTTP-методов, допустимых для маршрута, и функцию обработки запроса. В данном случае для обоих маршрутов допустимым считается только HTTP-метод GET. Чтобы добавить поддержку HTTP-метода DELETE (или любых других), достаточно определить функцию обработки такого запроса и добавить соответствующую информацию в вызов функции merge. Ниже приводится пример, где новый код выделен жирным:

```
let delValue request =
     new HttpResponseMessage() |> async.Return

let app = merge [route "/api/values" <| get values
                 route "/api/values/{id}" <| (get value <|> delete delValue)]
```

Одно из важнейших преимуществ библиотеки Frank – возможность использования приема композиции функций. Она позволяет легко добавлять новые функциональные возможности. В качестве примера я покажу, как добавить поддержку журналирования во все обработчики запросов. Следующий пример использует для этих целей одну из промежуточных функций, входящих в библиотеку Frank:

```
let app = merge [ route "/api/values" <| get values
                  route "/api/values/{id}" <| get value]
          |> Middleware.log
```

Единственная новая строка в этом примере включает простейшую поддержку журналирования трассировки всех запросов, включая продолжительность их обработки. Библиотека Frank предоставляет несколько таких функций, но вы легко можете создавать собственные промежуточные функции. Как это сделать, описывается на домашней странице проекта Frank, на сайте GitHub (https://github.com/frank-fs/frank).

Добавление перенаправления привязки

Так как библиотека Frank предназначена для использования из F# 2.0, при использовании ее из версии F# 3.0 требуется выполнить один дополнительный шаг. Вы должны определить перенаправление привязки из предыдущей версии сборки `FSharp.Core` в сборку `FSharp.Core` версии 4.3.0.0. Безусловно, это можно сделать вручную, но NuGet предоставляет более простое решение. Добавьте конфигурационный файл в проект и затем выполните следующую команду в консоли диспетчера пакетов NuGet (которую можно открыть, выбрав в Visual Studio пункт меню **View → Other Windows → Package Manager Console** (Вид → Другие окна → Консоль диспетчера пакетов)), заменив `projectname` именем своего проекта:

```
Add-BindingRedirect projectname
```

Она обновит конфигурационный файл, включив в него примерно такие строки:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="FSharp.Core"
          publicKeyToken="b03f5f7f11d50a3a" culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-4.3.0.0"
          newVersion="4.3.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Это необходимо сделать для любого проекта на F# 3.0, где используются библиотеки, собранные с предыдущей версией F#, поэтому держите эту команду под рукой.

Законченное решение, содержащее этот пример, можно взять на моей странице на сайте GitHub: https://github.com/dmohl/fs-web-cloud-mobile.

Тестирование своих творений

В первой главе я говорил о важности создания приложений, простых для тестирования. В нескольких разделах этой главы описывались различные приемы, позволяющие упростить тестирование кода на F#. Однако я не погружался в описание деталей создания модульных тестов на языке F# и/или каких-то его особенностей, способных упростить этот процесс.

В этом разделе я познакомлю вас с инструментами и приемами модульного тестирования своих шедевров на F#. Я также представлю несколько фреймворков, способных сделать тестирование более простым и приятным.

Подготовка

В качестве примера в следующих нескольких разделах я буду использовать контроллер ASP.NET Web API и репозиторий, очень по-

хожие на те, что вы видели в примере приложения ASP.NET MVC, в главе 1. Ниже приводится реализация контроллера:

```
namespace FsWeb.Controllers

open System
open System.Web.Http
open Microsoft.FSharp.Data.TypeProviders
open FsWeb.Models
open Repository

type dbContext = SqlDataConnection<ConfigFile="Web.config",
                                   ConnectionStringName="FsMvcAppExample">
type DataContext =
          dbContext.ServiceTypes.SimpleDataContextTypes.FsMvcAppExample

type GuitarsController(context:IDisposable, ?repository) =
     inherit ApiController()

     let fromRepository =
          match repository with
          | Some v -> v
          | _ -> (context :?> DataContext).Guitars |> get

new() = new GuitarsController(dbContext.GetDataContext())

// GET /api/guitars
member x.Get() =
     getAll() |> fromRepository |> List.toSeq

override x.Dispose disposing =
     context.Dispose()
     base.Dispose disposing
```

Реализация репозитория несколько проще той, что мы видели выше:

```
module Repository

open System
open System.Linq

let get (source:IQueryable<_>) queryFn =
     queryFn source |> Seq.toList
```

```
let getAll () =
     fun s -> query { for x in s do
                      select x }
```

Эти контроллер и репозиторий взаимодействуют с базой данных, которая так же была создана в первой главе. Поскольку база данных уже существует, для работы с ней я буду использовать поставщика типов `SqlDataConnection`.

Еще одна особенность F#, о которой я еще не упоминал и которая дает дополнительные удобства, – это *аббревиатуры типов* (type abbreviations). Поддержка аббревиатур типов, таких как `DataContext` в примере выше, позволяет определять псевдонимы типов, чтобы повысить удобочитаемость кода. Аббревиатуры типов могут пригодиться во многих ситуациях, но они особенно полезны при работе с типами, сгенерированными поставщиками типов.

Теперь вы сможете быстро написать тест, который проверит правильность данных, возвращаемых методом `Get` контроллера `GuitarsController`. Если вы включили тестовый проект при создании решения ASP.NET Web API, вы сможете добавить в него ссылки на свой фреймворк тестирования. В следующих примерах я буду использовать разные фреймворки. Так, в первом примере используется MSTest, а чтобы его подключить, достаточно добавить ссылку на сборку `Microsoft.VisualStudio.QualityTools.UnitTestFramework`.

> **Примечание.** Шаблон проекта F# MSTest можно найти в галерее Visual Studio Gallery или с помощью инструмента поиска шаблонов в Интернете в Visual Studio 2012.

Теперь, когда тестовый проект готов к использованию, добавьте в него следующий тест:

```
module GuitarsControllerTests

open Microsoft.VisualStudio.TestTools.UnitTesting
open FsWeb.Controllers

[<TestClass>]
type Tests() =
     [<TestMethod>]
     member x.GetAllGuitarsTest() =
          use controller = new GuitarsController()
          let result = controller.Get()
          Assert.AreEqual(3, result |> Seq.length)
```

Этот тест создаст контроллер, вызовет метод `Get` и сравнит возвращаемое им значение с ожидаемым числом 3.

Возможно, вас уже переполняют идеи, как можно улучшить этот тест. Возможно, вы думаете, что этот тест недостаточно удобочитаем, не изолирован, будет слишком медлительным из-за необходимости взаимодействовать с базой данных, и так далее. Думающие так абсолютно правы. Такой тест может быть и не плох для интеграционного тестирования, но он малопригоден в роли модульного теста. В следующих нескольких подразделах я познакомлю вас с некоторыми приемами, библиотеками, фреймворками и особенностями F#, которые помогут улучшить этот тест.

Улучшение тестов с применением F#

Как с применением F# можно улучшить только что написанный тест? Для начала познакомимся с особенностями языка, которые позволят улучшить удобочитаемость кода, а также переместим код в более удачное место:

```
module ``Given a Guitars Controller``

open Microsoft.VisualStudio.TestTools.UnitTesting
open FsWeb.Controllers

[<TestClass>]
type ``When getting all the guitars from the repository``() =

     let controller = new GuitarsController()
     let result = controller.Get()

     [<TestMethod>]
     member x.``it should have a count of three``() =
          Assert.AreEqual(3, result |> Seq.length)
```

В языке F# имеется одна замечательная особенность, которая называется *двойные обратные апострофы* (``) и позволяет включать в идентификаторы некоторые символы, недопустимые в обычном случае. С ее помощью можно определять весьма говорящие имена модулей, классов и методов. В предыдущем примере эта особенность применяется для включения пробелов в имена модуля, типа и метода. Она часто будет применяться в примерах, следующих ниже, при работе с различными библиотеками и фреймворками.

Другой особенностью F#, которая позволит не обращаться к базе данных и сделать тест более изолированным, являются *выражения объектов* (object expressions). Выражения объектов дают возможность создать объект с интерфейсом другого объекта, служащего в качестве своеобразного шаблона. После этого вы сможете переопределить некоторые аспекты нового объекта.

Если вы подумали, что речь идет о фиктивных объектах, то вы совершенно правы. Ниже приводится измененная версия примера (новый код выделен жирным), где выражение объекта используется для создания фиктивного репозитория:

```
module ``Given a Guitars Controller``

open System
open System.Linq
open Microsoft.VisualStudio.TestTools.UnitTesting
open FsWeb.Controllers
open FsWeb.Models

type Guitar = dbContext.ServiceTypes.Guitars

[<TestClass>]
type ``When getting all the guitars from the repository``() =

    let fakeRepository =
        [| Guitar(Id = Guid.NewGuid(), Name = "test1")
           Guitar(Id = Guid.NewGuid(), Name = "test2")
           Guitar(Id = Guid.NewGuid(), Name = "test3") |]
        |> Queryable.AsQueryable
        |> Repository.get

    let context = { new Object()
                        interface IDisposable with
                            member x.Dispose() = () }

    let controller = new GuitarsController(context, fakeRepository)
    let result = controller.Get()

    [<TestMethod>]
    member x.``it should have a count of three``() =
        Assert.AreEqual(3, result |> Seq.length)
```

Первый фрагмент кода создает список фиктивных данных для нужд тестирования. Затем список преобразуется в нечто, к чему

можно обратиться и передать функции `Repository.Get`. Когда позднее это нечто будет передано контроллеру `GuitarsController`, оно будет интерпретироваться как данные, полученные из базы данных с помощью поставщика типов `SqlDataConnection`.

Второй интересный момент, на который следует обратить внимание, – как обеспечивается поддержка интерфейса `IDisposable` в объекте `context`. Поскольку в фиктивном объекте репозитория отсутствует поддержка интерфейса `IDisposable`, мы должны добавить ее. Это легко сделать с помощью выражения объекта, создающего анонимный тип `Object` и реализующий `IDisposable` с методом `Dispose`, который ничего не делает.

Многие фреймворки позволяют использовать преимущества синтаксиса и других особенностей языка F#. В следующих нескольких разделах мы коротко исследуем некоторые из них.

FsUnit

Фреймворк FsUnit был создан Реем Вернагусом (Ray Vernagus) в 2007 году. С тех пор в его развитии принимали участие многие разработчики. В него была включена поддержка многих наиболее популярных фреймворков тестирования, включая xUnit.NET, MbUnit, изначально поддерживавшийся NUnit и совсем недавно добавленый MSTest.

Самый простой способ приступить к использованию фреймворка FsUnit – установить один из доступных NuGet-пакетов. Например, чтобы использовать FsUnit в сочетании с MSTest, установите пакет `Fs3OUnit.MsTest`. Он включает все необходимые библиотеки, добавляет ссылки и содержит простой пример его применения.

После установки фреймворка вы сможете переписать метод, используя более читаемый синтаксис, чем позволяет подход на основе утверждений `Assert`. Благодаря фреймворку FsUnit наш единственный тестовый метод превращается в:

```
[<TestMethod>]
member x.``it should have a count of three``() =
     result |> Seq.length |> should equal 3
```

Определив значение для одного из элементов последовательности с результатами, мы сможем создать тест, проверяющий его присутствие:

```
[<TestMethod>]
member x.``it should contain a specific guitar record``() =
     result |> should contain expectedGuitar
```

Как упоминалось выше, фреймворк FsUnit также поддерживает NUnit, MbUnit и xUnit.NET. Одной из самых приятных сторон этих трех фреймворков является возможность создавать тесты в функциональном стиле. Ниже приводится пример того же теста, написанного для FsUnit с xUnit.NET:

```
module ``When getting all guitars``

open System
open System.Linq
open FsWeb.Controllers
open FsWeb.Models
open FsUnit.Xunit
open Xunit

type Guitar = dbContext.ServiceTypes.Guitars

let expectedGuitar = Guitar(Id = Guid.NewGuid(), Name = "test1")

let fakeRepository =
     [| expectedGuitar
        Guitar(Id = Guid.NewGuid(), Name = "test2")
        Guitar(Id = Guid.NewGuid(), Name = "test3") |]
     |> Queryable.AsQueryable
     |> Repository.get

let context = { new Object()
                    interface IDisposable with
                        member x.Dispose() = () }

let controller = new GuitarsController(context, fakeRepository)
let result = controller.Get()

[<Fact>]
let ``it should have a count of three``() =
     result |> Seq.length |> should equal 3

[<Fact>]
let ``it should contain a specific guitar record``() =
     result |> should contain expectedGuitar
```

В арсенале этого фреймворка имеется несколько ключевых слов, позволяющих максимально повысить удобочитаемость тестов. Полный их перечень, а также различные примеры их использования можно найти по адресу: https://github.com/dmohl/fsunit.

Примечание. Как я уже упоминал выше в этой главе, библиотеки, предназначенные для использования из F# 2.0, требуют перенаправления привязки в сборку `FSharp.Core` версии 4.3.0.0. Это необходимо сделать, если FsUnit, NUnit, MbUnit и xUnit.NET используются из F# 3.0. Это так же следует сделать для опробования примеров использования Unquote и NaturalSpec, о которых я расскажу ниже.

Unquote

Unquote – еще одна замечательная библиотека для создания модульных тестов на F#. Она была написана Стивеном Свенсеном (Stephen Swensen). Поддерживает все фреймворки тестирования, опирающиеся на исключения, такие как xUnit.NET, NUnit, MbUnit и MSTest; совместима с интерактивной оболочкой F# Interactive, а также со многими традиционными приемами; и основана на использовании цитируемых выражений (quoted expressions) F#, с помощью которых обеспечивает возможность статической проверки. Предоставляет несколько удобных операторов, способных немного упростить вашу жизнь как разработчика. Кроме того, ясные и хорошо отформатированные сообщения об ошибках делают эту библиотеку отличным выбором.

Чтобы начать использовать Unquote, достаточно лишь установить NuGet-пакет `Unquote`, а также пакет какого-либо фреймворка тестирования. После этого вы сможете создавать тесты внутри проекта или в интерактивной оболочке FSI. Следующий пример демонстрирует, как можно протестировать контроллер `GuitarsController` с помощью Unquote и xUnit.NET:

```
module ``Given a GuitarsController``

open System
open System.Linq
open FsWeb.Controllers
open FsWeb.Models
open Swensen.Unquote
open Xunit
```

```
type Guitar = dbContext.ServiceTypes.Guitars

let fakeRepository =
    [| Guitar(Id = Guid.NewGuid(), Name = "test1")
       Guitar(Id = Guid.NewGuid(), Name = "test2")
       Guitar(Id = Guid.NewGuid(), Name = "test3") |]
    |> Queryable.AsQueryable
    |> Repository.get

let context = { new Object()
                    interface IDisposable with
                        member x.Dispose() = () }

let controller = new GuitarsController(context, fakeRepository)

[<Fact>]
let ``When getting the guitars from the repo it should have a count of 3``()
    = test <@ controller.Get() |> Seq.length = 3 @>
```

Основное отличие этого примера заключено в использовании оператора проверки утверждения `test`, за которым следует цитируемое выражение (quoted expression). Цитируемое выражение содержит код, который должен быть декомпилирован, выполнен и свернут. Свертка выражений и подвыражений выполняется рекурсивно, пока не будет возбуждено исключение или пока тест не будет выполнен до конца. Поскольку библиотека Unquote в первую очередь основана на цитируемых выражениях, для ее использования вам будет вполне достаточно знаний языка F#, которые у вас уже имеются и запомнить всего три оператора проверки утверждений: `test`, `raise` и `raiseWith`.

Саму библиотеку Unquote, документацию к ней, примеры, примечания к выпуску и другую информацию можно получить по адресу: http://code.google.com/p/unquote/.

NaturalSpec

Несмотря на наличие большого количества замечательных фреймворков тестирования для F#, ради экономии места в книге я познакомлю вас лишь с еще одним инструментом. NaturalSpec – это *предметно-ориентированный язык* (Domain-Specific Language, DSL), написанный Стефеном Форкманом (Steffen Forkmann) поверх NUnit. Как можно заключить из названия, NaturalSpec позволяет

создавать тесты, или спецификации, описывая их на естественном языке, практически так же, как вы могли описать критерии приемки. Он широко использует прямой конвейерный оператор, а также использует функции весьма интересным способом.

Следующий пример снова представляет тест для нашего контроллера `GuitarsController`. Новый код для работы с NaturalSpec выделен жирным:

```
module GuitarsControllerTests

open System
open System.Linq
open FsWeb.Controllers
open FsWeb.Models
open NaturalSpec

type Guitar = dbContext.ServiceTypes.Guitars

let expectedGuitar = Guitar(Id = Guid.NewGuid(), Name = "test1")

let fakeRepository =
    [| expectedGuitar
       Guitar(Id = Guid.NewGuid(), Name = "test2")
       Guitar(Id = Guid.NewGuid(), Name = "test3") |]
    |> Queryable.AsQueryable
    |> Repository.get

let context = { new Object()
                    interface IDisposable with
                        member x.Dispose() = () }

let a_GuitarController = new GuitarsController(context, fakeRepository)

let getting_all_guitars (c:GuitarsController) = c.Get()

[<Scenario>]
let ``When getting all the guitars from the repository``() =
    Given a_GuitarController
         |> When getting_all_guitars
         |> It should have (length 3)
         |> It should contain expectedGuitar
         |> Verify
```

В этом примере сначала создается экземпляр контроллера `GuitarsController`, а затем он передается функции `getting_all_guitars`. Ее результат передается далее каждой функции-утверждению. А в конце полученные результаты проверяются.

NaturalSpec поддерживает несколько способов компоновки ваших тестов и включает функции, дающие возможность улучшать читаемость тестов. Кроме того, для ситуаций, когда выражений объектов может оказаться недостаточно, поддерживается фреймворк создания фиктивных объектов. Это особенно удобно, когда NaturalSpec используется для тестирования традиционного объектно-ориентированного кода.

Примечание. NaturalSpec так же требует перенаправления привязки, о которой несколько раз упоминалось в этой главе. Следует отметить, что фреймворк NUnit не будет загружать конфигурационный файл, если не следовать определенным соглашениям об именовании. На практике я обнаружил, что вполне достаточно инструкций, связанных с единственным сценарием `AppDomain`.

В заключение

В этой главе вы познакомились с различными способами создания и тестирования служб на языке F#. Эти службы могут опираться на различные фреймворки и решать самые разные задачи. Независимо от того, какой фреймворк будет выбран, уникальные особенности F# помогут обогатить ваш опыт.

В следующей главе мы рассмотрим возможность использования F# в сочетании с платформой Windows Azure. Я покажу, как использовать эту комбинацию для создания еще более надежных и масштабируемых решений, и как можно применить знания, уже имеющиеся у вас, для создания облачных приложений на F#. Кроме того, я продолжу знакомить вас с новыми особенностями языка F#, а также инструментами, фреймворками и приемами. Вперед и вверх!

Глава 3. К облакам! Использование преимуществ Azure

Будущее уже здесь. Только оно неравномерно распределено.
– Уильям Гибсон (William Gibson)

В предисловии к этой книге упоминалось, что основным направлением развития технологий является создание эффективных, масштабируемых и высоко доступных приложений, способных выполняться на любых современных вычислительных устройствах, имеющихся у пользователей. Одним из основных инструментов достижения этой цели является платформа Windows Azure. За облачными технологиями будущее, и такие платформы, как Windows Azure помогают равномерно распределить его.

Преимущества Windows Azure трудно не заметить. Затраты на запуск и эксплуатацию ниже, чем при использовании локальной системы, вам не придется беспокоиться о поддержании аппаратных средств и уходе за ними, и вы сможете использовать большую часть своих навыков разработки. С другой стороны, существует несколько важных отличий, которые необходимо принять во внимание при проектировании и создании облачных приложений. В частности, масштабирование облачных приложений выполняется за счет роста количества серверов (горизонтальное масштабирование), а не за счет увеличения производительности отдельных элементов системы (вертикальное масштабирование). F# отлично подходит для создания приложений подобного типа.

В этой главе проведу вас через примеры создания и развертывания облачных приложений на F#. Я также покажу, как взаимодействовать с другими ролями в вашем приложении и как использовать некоторые из особенностей Windows Azure. Наконец, я представлю

несколько замечательных библиотек и фреймворков для F#, созданных специально для использования преимуществ Windows Azure.

Прежде чем погрузиться в обсуждение, я должен сделать пару замечаний относительно данной главы. Во-первых, я настоятельно рекомендую познакомиться с пошаговыми руководствами на странице http://www.windowsazure.com/en-us/develop/net/[1], описывающими способы взаимодействий с платформой Windows Azure посредством Windows Azure SDK для .NET. Вместо того, чтобы повторять содержимое этих руководств в данной главе, я буду давать лишь самые важные выдержки из них, а также описывать особенности реализации, имеющие непосредственное отношение к языку F#. Желающим получить дополнительную информацию о платформе Windows Azure я могу порекомендовать прочитать книгу «Programming Windows Azure», написанную Шрирамом Кришнаном (Sriram Krishnan) (O'Reilly).

Во-вторых, в данной главе предпринят несколько иной подход, чем в двух предыдущих главах. Первые несколько разделов знакомят с множеством понятий и приемов взаимодействий с Windows Azure. Хотя обсуждение сопровождается некоторыми демонстрационными примерами, но они носят весьма абстрактный характер. Если вы ожидаете познакомиться с более практичными примерами, не пугайтесь. В разделе «Создание масштабируемых приложений» будет представлен более реальный пример, в котором применяются некоторые абстрактные аспекты из предыдущих примеров, а также демонстрирует практические приемы создания приложений для платформы Windows Azure.

Создание и развертывание приложений F# на платформе Azure

Платформа Windows Azure предлагает превосходный комплект инструментов для создания и развертывания приложений в Веб. Для начала просто загрузите и установите пакет Windows Azure SDK для .NET, который можно получить на сайте Windows Azure .NET Developer Center (https://www.windowsazure.com/en-us/develop/net/).

---

[1] Похожие руководства на русском языке можно найти на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/windowsazure/dd179367.aspx. – *Прим. перев.*

Примечание. Поддержка Windows Azure SDK для .NET была добавлена в версии Visual Studio 2012 RC в июне 2012. Примеры в данной книге были проверены с использованием этой версии.

Как упоминалось выше, при создании приложений для Windows Azure вы можете использовать все знания и навыки, полученные в этой книге. Например, чтобы создать и развернуть приложение на основе шаблона проекта F#/C# ASP.NET Web API, рассматривавшегося в главе 2, достаточно выполнить лишь несколько следующих шагов:

1. Запустите Visual Studio от имени пользователя Administrator (Администратор).
2. Создайте решение F#/C# ASP.NET Web API, выбрав установленный шаблон **Visual F# → ASPNET → F# and C# Web Application (ASP.NET MVC 4)**.
3. Добавьте в решение новый проект Windows Azure, создав его из шаблона **Visual C# → Cloud → Windows Azure Cloud Service** (Visual C# → Cloud → Облачная служба Windows Azure). Щелкните на кнопке **OK** после заполнения параметров проекта.

Внимание. У вас может появиться искушение выбрать шаблон ASP.NET MVC 4 Web Role (Веб-роль ASP.NET MVC 4) в окне мастера создания проекта Windows Azure, но не поддавайтесь ему.

4. Наконец, щелкните правой кнопкой на папке *Roles (Роли)* в новом проекте и добавьте веб-приложение, выбрав пункт контекстного меню **Web Role Project in solution...** (Проект веб-роли в решении)

Вот и все. Панель **Solution Explorer** (Обозреватель решений) должна выглядеть примерно так, как показано на рис. 3.1. Теперь можно запустить приложение локально, с помощью Windows Azure Compute Emulator.

Примечание. Если в вашей среде разработки не установлен компонент SQL Express, при попытке запустить приложение локально вы получите сообщение о неудачной попытке инициализировать эмулятор хранилища. Чтобы избавиться от этой проблемы, эмулятору хранилища, что он должен использовать экземпляр DB. Сделать это можно, запустив Windows Azure из командной строки и введя команду `DSInit /sqlinstance:<имя экземпляра>` (например, `DSInit /sqlinstance:SQL2K8`).

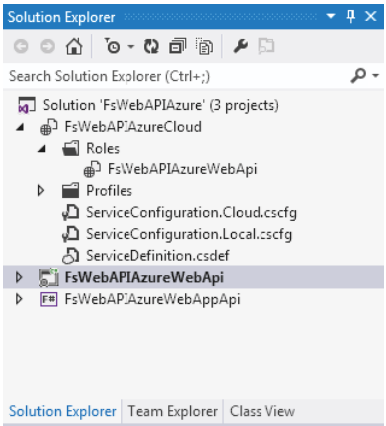

Рис. 3.1. Решение F# ASP.NET Web API Azure

Теперь это решение можно поместить в облако, следуя инструкциям в разделе «Deploy the Application to Windows Azure» (Развертывание приложения в Windows Azure), в электронном руководстве по адресу http://bit.ly/windows-azure. Как отмечалось во введении к этой главе, я не буду повторять информацию, содержащуюся в различных электронных руководствах, поэтому, если прежде вам не приходилось заниматься развертыванием приложений на платформе Windows Azure, я настоятельно рекомендую прочитать данное руководство.

Создание рабочей роли на F#

Создать рабочую роль на F# еще проще, чем создать веб-роль. Эта простота в значительной степени обусловлена включением шаблона рабочей роли на F# в комплект Windows Azure Tools for Microsoft Visual Studio (составная часть пакета SDK, упомянутого выше), который прежде требовалось загружать отдельно. Для этого нужно лишь добавить в решение новый проект Windows Azure из шаблона, который можно найти, выбрав **Visual C# → Cloud → Windows Azure Cloud Service** (Visual C# → Cloud → Облачная служба Windows Azure), в появившемся окне щелкните дважды на шаблоне Visual F# Worker Role, измените имя проекта рабочей роли (если необходимо) и щелкните на кнопке **OK**. На рис. 3.2 показано окно мастера

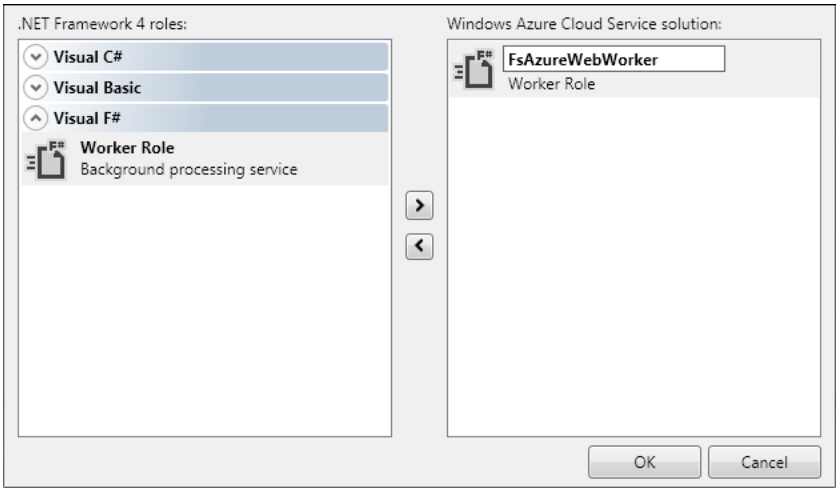

Рис. 3.2. Окно мастера создания проекта Windows Azure после выбора шаблона F# Worker Role (Рабочая роль F#) и изменения имени проекта

создания проекта Windows Azure после выбора шаблона **F# Worker Role** (Рабочая роль F#) и изменения имени проекта.

Введение в библиотеку Fog

Платформа Windows Azure экспортирует свой API посредством HTTP-служб. Это очень выгодно, потому что позволяет использовать любые языки программирования для реализации взаимодействий с этой платформой. Кроме того, были созданы различные обертки, еще больше упрощающие организацию взаимодействий на отдельных языках. При создании приложений для .NET, проще всего использовать комплект инструментов Windows Azure SDK для .NET, упоминавшийся выше.

SDK для .NET можно использовать в приложениях на F# практически так же, как в приложениях на C# или VB.NET, однако этот комплект инструментов в значительной степени ориентирован на использование объектно-ориентированной парадигмы. Чтобы получить возможность использовать некоторые уникальные особенности F#, я создал библиотеку Fog.

Библиотека Fog спускает облако на землю и дает возможность использовать его из F#. Цель библиотеки – уменьшить объем типового

кода, а также обеспечить возможность реализации взаимодействий с Windows Azure в более функциональном стиле. На протяжении оставшейся части главы я буду приводить фрагменты кода, которые вы сможете использовать для взаимодействий с платформой Azure без библиотеки Fog, а также покажу, насколько проще может быть ваша жизнь, благодаря этой библиотеке. Получить библиотеку Fog можно на моей странице, на сайте GitHub (https://github.com/dmohl/Fog), или загрузить ее как NuGet-пакет с идентификатором `Fog`.

Взаимодействие с хранилищами данных Azure

Платформа Windows Azure поддерживает несколько разных служб хранения данных. Вероятнее всего вы будете пользоваться службой таблиц и SQL Azure, когда потребуется поддержка транзакций, и службой хранения больших двоичных объектов (Blob) для хранения больших порций данных, таких как файлы. В зависимости от потребностей, вам может также пригодиться служба очередей.

В этом разделе я расскажу, как сохранять и извлекать данные с помощью этих служб хранения, используя F# и Windows Azure .NET API. Кроме того, я покажу, как использовать для этой цели библиотеку Fog.

Большие двоичные объекты

Служба хранения больших двоичных объектов является основным средством хранения объемных типов данных. На каждую учетную запись в службе хранения больших двоичных объектов отводится примерно 100 Тбайт пространства. Хранилище двоичных объектов может быть разделено на несколько логических контейнеров и обычно используется для хранения неструктурированных данных, таких как изображения, видео- и аудиозаписи, и файлов других типов. На веб-сайте Windows Azure (https://www.windowsazure.com/en-us/develop/net/how-to-guides/blob-storage/) можно найти замечательное руководство, описывающее, как получить и задействовать хранилище двоичных объектов.

С точки зрения программирования на языке F#, вы могли бы практически «в лоб» перевести примеры из упомянутого руководства на язык F#, добавить этот код в веб- или рабочую роль и за-

кончить на этом. Ниже приводится пример, как можно реализовать обращение к API для создания двоичного объекта:

```
let storageAccount =
     RoleEnvironment.GetConfigurationSettingValue "BlobStorageConnectionString"
     |> CloudStorageAccount.Parse

let client = storageAccount.CreateCloudBlobClient()
let container = client.GetContainerReference("testcontainer")
container.CreateIfNotExist() |> ignore

let blob = GetBlobReferenceInContainer container "testblob"
blob.UploadText "My super awesome text to upload"
```

Для организации взаимодействий со службой хранения больших двоичных объектов с помощью библиотеки Fog, установите NuGet-пакет `Fog` в веб- или рабочую роль Azure. Затем добавьте строку подключения к службе с именем `BlobStorageConnectionString`. После этого вы сможете выгрузить двоичный объект всего одной строкой кода:

```
UploadBlob "testcontainer" "testblob" "My super awesome text to upload"
```

Извлечь этот объект можно следующим образом:

```
DownloadBlob<string> "testcontainer" "testblob"
```

Вы можете также удалять двоичные объекты, удалять контейнеры и получать/устанавливать метаданные двоичных объектов. Для тех, кого не устраивают настройки по умолчанию, в библиотеке Fog имеются функции, более точно соответствующие методам в Windows Azure SDK для .NET. Они обеспечивают настолько высокую точность управления различными аспектами, насколько может потребоваться. Например, чтобы извлечь двоичный объект с той же точностью, как было показано в примере, полученного прямым переводом программного кода с языка C# на F#, можно следующим образом:

```
let container = GetBlobContainer <| BuildBlobClient() <| "testcontainer"
UploadBlobToContainer container "testblob" "This is a test"
```

Возможно, вам потребуется использовать строку подключения с другим именем. Вы можете добиться этого, как показано ниже:

```
let container = GetBlobContainer
                <| BuildBlobClientWithConnStr "SomeOtherConnStr"
                <| "testcontainer"
UploadBlobToContainer container "testblob" "This is a test"
```

Я не буду углубляться в детали реализации Fog, но мне хотелось бы указать на некоторые особенности языка F#, которые пригодились мне при создании этой библиотеки. Первая интересная особенность, на которую я хочу обратить ваше внимание, – как F# может предупредить вас при работе с классами, реализующими интерфейс `IDisposable`, и потому должны использоваться с ключевыми словами `use` или `using`. F# не требует указывать ключевое слово `new` при создании экземпляров классов, поэтому я часто опускаю его. Однако в отсутствие ключевого слова `new` выводится предупреждение при использовании класса, реализующего интерфейс `IDisposable`. Оно может служить отличным индикатором и напоминанием, что вы должны использовать `use` или `using`, а также ключевое слово `new` для создания экземпляров этого класса. В процессе работы над библиотекой Fog, эта особенность языка F# помогала быстро находить места, где требовалось явное освобождение объектов.

Вторая особенность – возможность использования оператора сопоставления с образцом для проверки типа значения – оказалась весьма полезной для реализации функций выгрузки и загрузки двоичных объектов. В состав Windows Azure SDK для .NET входит множество разных методов для загрузки и выгрузки данных различных типов. Поддержка возможности сопоставления с образцами типов позволила мне написать единственную функцию, обрабатывающую все поддерживаемые типы. Выражение сопоставления для функции выгрузки выглядит, как показано ниже:

```
match box item with
| :? Stream as s -> blob.UploadFromStream s
| :? string as text -> blob.UploadText text
| :? (byte[]) as b -> blob.UploadByteArray b
| _ -> failwith "This type is not supported"
```

Таблицы

Служба Azure хранения таблиц – это нереляционное хранилище структурированных и частично структурированных данных. Служба хранения таблиц – отличное решение для хранения данных в формате, позволяющем быстро извлекать их и отображать на сайте. Од-

ним из типичных примеров может служить использование службы таблиц для ненормализованного представления данных, отображаемых на сайте или в отчетах.

Службу хранения таблиц можно считать базой данных NoSQL, напоминающей Couchbase, CouchDB, MongoDB, RavenDB и другие. Одно из существенных отличий состоит в том, что служба таблиц в Azure обеспечивает немедленную целостность данных, а не в конечном итоге. Многие базы данных NoSQL обеспечивают целостность данных только в конечном итоге, поэтому операция считается успешной уже после обновления единственного узла. Это ускоряет работу, но при чтении из разных узлов потенциально могут возвращаться не самые последние данные. Служба таблиц гарантирует непротиворечивость данных между узлами до того, как завершится операция сохранения. Это несколько замедляет скорость работы, но гарантирует немедленную целостность.

Взаимодействия со службой таблиц с помощью Fog реализуются особенно просто, когда строка подключения в настройках определена с именем `TableStorageConnectionString`. Добавив такую строку, вы сможете использовать следующий код для добавления, изменения и удаления сущностей в хранилище таблиц Azure:

```
[<DataServiceKey("PartitionKey", "RowKey")>]
type TestClass() =
     let mutable partitionKey = ""
     let mutable rowKey = ""
     let mutable name = ""
     member x.PartitionKey with get() = partitionKey and set v = partitionKey <- v
     member x.RowKey with get() = rowKey and set v = rowKey <- v
     member x.Name with get() = name and set v = name <- v

let originalClass =
     TestClass(PartitionKey = "TestPart",
               RowKey = Guid.NewGuid().ToString(),
               Name = "test" )

CreateEntity "testtable" originalClass |> ignore

let newClass = originalClass
newClass.Name <- "test2"

UpdateEntity "testtable" newClass |> ignore

DeleteEntity "testtable" newClass
```

Примечание. Кроме случаев, где это отмечено явно, параметры настройки для библиотеки Fog должны храниться в файлах *.cscfg, находящихся в проекте Azure.

Обратите внимание на атрибут `DataServiceKey` и свойства с именами `PartitionKey` и `RowKey` в классе, экземпляры которого действуют как хранимые сущности. Они необходимы для организации взаимодействий со службой таблиц Windows Azure через Windows Azure SDK для .NET. Добиться того же эффекта, можно унаследовав класс `TableStorageEntity`.

Теперь, когда инструменты разработки для Windows Azure включены в состав Visual Studio 2012, можно использовать вычислительные выражения запросов, появившиеся в F# 3.0 и обсуждавшиеся в главе 1. Если вы пользуетесь Visual Studio 2010, лучшим способом выполнения запросов к службе таблиц Azure из F# является использование поддержки LINQ в пакете F# PowerPack. Установив NuGet-пакет `FSPowerPack.Linq.Community` и открыв в программе `Microsoft.FSharp.Linq.Query`, для извлечения конкретной сущности можно использовать следующий код:

```
query <@ seq { for e in context.CreateQuery<TestRecord>("testtable") do
                    if e.PartitionKey = testRecord.PartitionKey &&
                            e.RowKey = testRecord.RowKey then
                       yield e } @> |> Seq.head
```

Служба хранения очередей

В настоящее время, после появления поддержки Azure Service Bus, очереди используются не так часто, однако они обладают особенностями, которые могут пригодиться в некоторых ситуациях. Одной из таких ситуаций является необходимость хранения сообщений, общим объемом более 5 Гбайт. В библиотеке MSDN можно найти полный перечень сценариев, где вместо Azure Service Bus следует использовать службу очередей.

Библиотека Fog делает работу со службой очередей Azure такой же простой, как и с другими службами хранения, описанными выше. Просто добавьте строку подключения с именем `QueueStorageConnectionString` и используйте следующий код для добавления, извлечения и/или удаления сообщения:

```
AddMessage "testqueue" "This is a test message" |> ignore
let result = GetMessages "testqueue" 20 5
for m in result do
     DeleteMessage "testqueue" m
```

В реализации взаимодействий со службой очередей, а также в некоторых других местах внутри библиотеки Fog, был использован один из приемов функционального программирования, который называется *мемоизация* (memoization). Прием мемоизации часто используется для кеширования результатов, возвращаемых функциями, чтобы ускорить их выполнение. В библиотеке Fog используется функция `memoize`, как описывается в статье Дона Сайма (Don Syme)[1]. Следующий фрагмент демонстрирует применение этого приема:

```
memoize (fun conn ->
             let storageAccount = GetStorageAccount conn
             storageAccount.CreateCloudQueueClient() ) connectionString
```

Этот код создает клиента облачной очереди для использования всеми другими функциями, работающими со службой очередей Azure в библиотеке Fog. Благодаря применению `memoize`, создание клиента выполняется только один раз и в ответ на все остальные запросы просто возвращается ранее созданный клиент.

SQL Azure

Служба SQL Azure поддерживает облачный экземпляр SQL практически с такой же функциональность, как и сервер MS SQL, с которым вы наверняка хорошо знакомы. Организация взаимодействий со службой SQL Azure из F# мало отличается от взаимодействий с MS SQL. Поэтому я не буду тратить много времени на эту тему и лишь укажу на несколько примеров в электронных руководствах.

Существует несколько способов взаимодействий с MS SQL из F#. Конкретный пример использования SQL Azure можно найти в статье Мэттью Молони (Matthew Moloney)[2]. Томас Петричек (Tomas Petricek) написал замечательную статью, описывающую чтение дан-

[1] http://blogs.msdn.com/b/dsyme/archive/2007/05/31/a-sample-of-the-memoization-pattern-in-f.aspx.

[2] http://blog.moloneymb.com/?p=206.

ных из базы данных SQL[1]. Маурисио Шеффер (Mauricio Scheffer) написал превосходную библиотеку FsSql, являющуюся функциональной оберткой вокруг ADO.NET, которую можно найти на сайте GitHub (https://github.com/mausch/FsSql). Существует множество других способов взаимодействий с MS SQL из F#, включая те, что я показывал в главах 1 и 2. Далее в этой главе я представлю еще один пример взаимодействий со службой SQL Azure с помощью Entity Framework 5.

Использование преимуществ Azure Service Bus

Инфраструктура Windows Azure Service Bus предоставляет удобный способ взаимодействий между узлами, ролями и/или приложениями в широкомасштабных распределенных решениях высокой доступности. Windows Azure Service Bus поддерживает также возможность ретрансляции сообщений, но в этом разделе я остановлюсь лишь на агентах передачи сообщений.

Очереди

Очереди в инфраструктуре Windows Azure Service Bus несколько напоминают очереди, о которых рассказывалось в разделе «Служба хранения очередей» выше. Однако очереди в инфраструктуре Service Bus лучше подходят для ситуаций, где требуется организовать обмен данными между службами. Как упоминалось выше, на сайте MSDN (http://msdn.microsoft.com/ru-ru/library/hh767287(VS.103).aspx.) приводится сравнительная характеристика этих двух технологий и дается все необходимая информация, которая поможет принять решение в выборе той или иной службы.

По традиции, сложившейся в этой главе, одним из самых простых способов использования очередей Azure Service Bus из F# является применение библиотеки Fog. В табл. 3.1 перечислены рекомендуемые настройки, позволяющие упростить использование Fog при работе с Azure Service Bus.

---

[1] http://tomasp.net/blog/dynamic-sql.aspx.

Таблица 3.1. Настройки Azure Service Bus для библиотеки Fog

Имя параметра	Описание
ServiceBusIssuer	Определяет значение параметра `Default Issuer`, которое можно получить, следуя инструкциям в разделе «Obtain the Default Management Credentials for the Namespace» (Получение параметров доступа по умолчанию для пространства имен), на веб-странице Windows Azure, описывающей очереди Service Bus (https://www.windowsazure.com/en-us/develop/net/how-to-guides/service-bus-relay/)
ServiceBusKey	Определяет значение параметра `Default Key`, которое можно найти там же, где и значение для параметра `Default Issuer`
ServiceBusScheme	Определяет значение первого аргумента метода `CreateServiceUri`
ServiceBusNamespace	Определяет имя пространства имен, которое определяется на шаге 4 в разделе «Create a Service Namespace» (Создание пространства имен службы), на веб-странице Windows Azure, описывающей очереди Service Bus (https://www.windowsazure.com/en-us/develop/net/how-to-guides/service-bus-relay/)
ServiceBusServicePath	Определяет значение, которое будет добавляться в конец URI. Может пригодиться при организации ретрансляции, чтобы определить выполняемую операцию. Для случаев применения агентов обмена сообщениями, это значение можно оставить пустым

Темы

Темы (topics) позволяют реализовать архитектуру издатель/подписчик, когда один ил более издателей отправляют сообщения, получаемые одним или более подписчиками. Когда выполняется подписка на тему, создается нечто похожее на виртуальную очередь для подписчика, благодаря которой он будет получать свои копии сообщений. Все остальные подписчики на тему так же будут получать свои копии сообщений. Такая схема обмена сообщениями может пригодиться в самых разных ситуациях.

Например, один из типичных приемов в приложениях, использующих хранилища данных NoSQL, заключается в том, чтобы сохранять ненормализованную версию (или версии) данных в хранилище

NoSQL и нормализованную версию – в хранилище SQL. Это позволяет достичь высокой производительности на стороне хранилища NoSQL, используя при этом особенности, предоставляемые хранилищами SQL, которые лучше подходят, например, для обработки данных и составления отчетов.

Реализовать это можно отправив единственное сообщение, которое будет извлекаться процессом и последовательно передаваться каждой службе хранения. Однако производительность такой реализации будет невысокой, потому что она будет вынуждена ждать, пока будет выполнено сохранение данных в каждом хранилище. Если позднее вам понадобится добавить дополнительные хранилища (например, хранилище метаданных), такая последовательная реализация сохранения еще больше увеличит продолжительность выполнения. Кроме того, добавление новых хранилищ данных может повлечь необходимость модификации того единственного процесса-подписчика, рассылающего запросы к хранилищам данных. Это может привести к образованию тесных связей между компонентами решения, препятствующей масштабируемости и возможности дальнейшего расширения.

С использованием модели издатель/подписчик, рабочие роли, ответственные за обновление конкретных хранилищ данных, могли бы подписаться на определенную тему. Тогда один или более издателей могли бы помещать в тему сообщения с новыми данными, а каждый подписчик мог бы извлекать свою копии сообщений и обновлять свое хранилище данных. Соедините это с возможностью асинхронных вызовов, и вы получите слабо связанное, быстрое решение, легко масштабируемое за счет добавления дополнительных серверов. В разделе «Создание масштабируемых приложений», ниже, я представлю пример использования такого подхода. А сейчас познакомимся с основами взаимодействий с темами Azure Service Bus.

Самый простой способ задействовать библиотеку Fog для организации взаимодействий с темами Azure Service Bus – добавить в конфигурационный файл параметры настройки, упоминавшиеся в разделе «Очереди», выше, и затем использовать следующий код:

```
type TestRecord = { Name : string }
let testRecord = { Name = "test" }

Subscribe "topictest" "AllTopics1"
```

```
    <| fun m -> printfn "%s" m.GetBody<TestRecord>().Name
    <| fun ex m -> raise ex

Subscribe "topictest" "AllTopics2"
    <| fun m -> printfn "%s" m.GetBody<TestRecord>().Name
    <| fun ex m -> raise ex

Publish "topictest2" testRecord
```

Выделенный фрагмент создает двух подписчиков, один – с именем `AllTopics1`, и второй – с именем `AllTopics2`. Каждый из них подписан на тему `topictest`. Затем в тему `topictest` посылается сообщение, представленное экземпляром записи с именем `testRecord`, в результате чего каждый из двух подписчиков получит сообщение.

Чтобы отписаться от темы, достаточно единственного вызова:

```
Unsubscribe "topictest" "AllTopics1"
```

Удаление темы так же выполняется единственным вызовом, как показано в следующем примере. Следует заметить, что большинство функций в библиотеке Fog проверяют факт создания перед удалением (на случай, если по каким-то причинам предпринята попытка удалить то, что не создавалось). Это означает, что если передать функции `DeleteTopic` имя несуществующей темы, это приведет к созданию новой темы с последующим ее удалением:

```
DeleteTopic "topictest"
```

Аутентификация и авторизация

Платформа Windows Azure включает несколько возможностей организации аутентификации и авторизации. Конечно, можно создать сайт, использующий механизм Membership Provider (поставщик участия) в Azure (с помощью пакета ASP.NET Universal Providers, который можно найти в NuGet под именем `System.Web.Providers`). Однако предпочтения в мире безопасности постепенно смещаются в сторону стандартных протоколов, таких как SAML и OAuth, обеспечивающих возможность *федеративной однократной регистрации* (federated single signon). Эти протоколы являются огромным шагом в правильном направлении, но они известны, как весьма сложные в настройке.

Платформа Azure превращает эту настройку в нечто тривиальное, предоставляя *службу управления доступом* (Access Control Service, ACS). Всего несколькими щелчками мыши в портале управления Azure (Azure Management Portal) можно настроить сайт на прием специальных маркеров (security tokens) от основных поставщиков идентичности (identity providers), таких как Google, Yahoo! и Windows Live. Пошаговые инструкции по этой настройке можно найти на веб-странице Windows Azure Access Control Service (http://bit.ly/bus-queues).

Приложив небольшие дополнительные усилия, можно добавить поддержку собственного поставщика идентичности, такого как экземпляр Active Directory Federation Services (ADFS) 2.0, способный аутентифицировать пользователей в каталоге Active Directory вашей организации. Пошаговые инструкции по настройке можно найти по адресу: http://msdn.microsoft.com/en-us/library/ff423674.aspx.

Данная слишком обширна, и ей было посвящено несколько книг, поэтому я не буду тратить слишком много времени на обсуждение деталей в этой книге. Вместо этого я познакомлю вас с некоторыми ключевыми понятиями и покажу пару способов работы с Windows Identity Foundation – основным компонентом платформы .NET, упрощающим реализацию федеративной аутентификации и авторизации – из F#. Дополнительную информацию об аутентификации и авторизации на основе заявок (claims-based) можно найти на веб-странице: http://msdn.microsoft.com/en-us/library/ff423674.aspx.

Аутентификация и авторизация с применением ACS

Хотя термины *аутентификация* и *авторизация* часто употребляются вместе, в действительности это две разные вещи. *Аутентификация* – это акт определения идентичности пользователя. *Авторизация* – это акт определения прав некоторого пользователя на доступ к определенному ресурсу. Таким ресурсом может быть все, что угодно, от всего сайта в целом, до отдельной веб-страницы или операции.

Служба управления доступом (Access Control Service, ACS) способна помочь реализовать и аутентификацию, и авторизацию. Она может также использоваться только для аутентификации и оставлять решение вопросов авторизации за вашим приложением. Наконец, может использоваться гибридный подход, когда служба ACS выполняет функции аутентификации и частично – авторизации,

а ваше приложение обеспечивает более точное управление авторизацией. Рассмотрим особенности реализации аутентификации и авторизации на нескольких примерах кода на F#.

Аутентификация на основе заявок

Следуя руководству по настройке аутентификации веб-пользователей с помощью службы ACS (https://www.windowsazure.com/en-us/develop/net/how-to-guides/access-control/[1]), можно настроить использование поставщиков идентичности Windows Live, Google и/или Yahoo!. Помимо того, что рассказывается в этом руководстве, при настройке веб-приложения на языке F# необходимо добавить ссылку на сборку `Microsoft.IdentityModel` и использовать код, как показано ниже, создающий строку представления всех типов заявок и связанных с ними значений:

```
let claimsIdentity = Thread.CurrentPrincipal.Identity :?> ClaimsIdentity
this.ViewData["Claims"] <-
     claimsIdentity.Claims
     |> Seq.fold(
          fun acc c -> acc +
               sprintf "Type: %s - Value: %s\r\n" c.ClaimType c.Value) ""
```

Эта информация не предназначена для конечного пользователя, но она будет весьма ценной для администратора. Чтобы вывести ее, можно добавить в реализацию представления следующий код:

```
<pre>@ViewData["Claims"]</pre>
```

Я добавил код этого примера в контроллер `HomeController` приложения ASP.NET Web API Azure, о котором рассказывалось в разделе «Создание и развертывание приложений F# на платформе Azure», выше. Этот код получает экземпляр `Identity` (описывающий идентичность пользователя) и приводит его к типу `ClaimsIdentity`, что упрощает нам задачу извлечение деталей заявки на создание специального маркера (security token). Далее код использует `Seq.fold`, чтобы создать единую строку для отображения на странице, чтобы показать каждый тип заявки и ее значение.

---

[1] Аналогичную статью на русском языке можно найти по адресу: http://blogs.msdn.com/b/natale/archive/2012/02/29/windows-azure.aspx. – *Прим. перев.*

При первом знакомстве функция `Seq.fold` может показаться кому-то странной. Она принимает коллекцию, выполняет обход ее элементов в цикле, добавляя их значения в аккумулятор (accumulator, часто сокращается до `acc`), и в конечном итоге возвращает значение этого аккумулятора. В данном примере функции `Seq.fold` передается коллекция заявок. В каждой итерации она добавляет в аккумулятор (строка в данном случае) текущий объект, представляющий заявку. Строковое представление типа заявки и ее значения добавляется в конец аккумулятора и после обработки всех заявок возвращается получившаяся строка аккумулятора. Две двойные кавычки в конце вызова `Seq.fold` определяют начальное значение аккумулятора. Возможно вы помните, что для реализации подобной функциональности в главе 2 использовался несколько иной синтаксис, на основе функции `Seq.reduce`.

Авторизация на основе заявок

Для нужд авторизации есть несколько вариантов на выбор. Можно использовать подход, похожий на тот, что был представлен в предыдущем разделе, и разрешать или запрещать доступ, опираясь на значение и/или присутствие определенной заявки. Например, чтобы лишь показать информацию о заявке, добавленной в предыдущем разделе, когда пользователь зарегистрирован в почтовой службе Gmail, можно было бы использовать следующий код:

```
let claimsIdentity = Thread.CurrentPrincipal.Identity :?> ClaimsIdentity

claimsIdentity.Claims
|> Seq.filter(fun c -> c.ClaimType = ClaimTypes.Email)
|> Seq.map(fun c -> c.Value)
|> Seq.head
|> function
   | v when v.Contains("@gmail.com") ->
       this.ViewData.["Claims"] <-
           claimsIdentity.Claims
           |> Seq.fold(
                  fun acc c -> acc +
                               sprintf "Type: %s - Value: %s\r\n"
                                   c.ClaimType c.Value) ""
   | _ -> ()
```

Выделенный фрагмент в этом примере получает последовательность заявок и выбирает из нее только заявки с типом, соответствую-

щим адресу электронной почты. Затем он преобразует последовательность оставшихся заявок в новую последовательность, содержащую только значения свойства `Value` каждого экземпляра `Claim`. Далее он возвращает первый результат из последовательности и выполняет сопоставление с образцом, чтобы определить, содержит ли значение `Value` подстроку `"@gmail.com"`.

Примечание. Если прежде вам не приходилось сталкиваться с функциями высшего порядка (higher-order functions) из модуля `Seq` и у вас есть опыт программирования на C#, просто представляйте себе `Seq.filter` как аналог метода `Enumerable.Where`, `Seq.map` – как аналог `Enumerable.Select`, `Seq.head` – как аналог `Enumerable.First`, и `Seq.fold` – как аналог `Enumerable.Aggregate`.

Второй способ основан на использовании соглашений, принятых в стеке *Windows Identity Foundation* (WIF), согласно которым экземпляры `Claim` преобразуются в роли. Этот прием открывает доступ к типичным методам авторизации, таким как `IsInRole`. Для этого стек WIF по умолчанию проверяет наличие заявки с типом http://schemas.microsoft.com/ws/2008/06/identity/claims/role. В мире ASP.NET MVC и ASP.NET Web API этот прием также открывает доступ к атрибуту `Authorize` а также к таим библиотекам, как Fluent Security, доступной в виде NuGet-пакета с именем `FluentSecurity`.

Создание масштабируемых приложений

К настоящему моменту я уверен, что вы уже заметили некоторые из преимуществ использования платформы Azure для создания масштабируемых приложений. Первое, что необходимо иметь в виду, создавая приложения на этой платформе, – они должны проектироваться с прицелом на горизонтальное масштабирование, а не на вертикальное. То есть, вы должны создавать приложения, которые могут распределяться на несколько серверов и работать над решением общей задачи. Многие из особенностей языка F#, обсуждавшихся выше, помогут вам в достижении этой цели.

В этом разделе я проведу вас через пример создания простого приложения для размещения заказов на комплекты игры в мешочек (hacky sacks). Приложение будет содержать одну веб-роль и две рабочие роли. В нем также будут использоваться некоторые возможности платформы Azure, с которыми мы уже познакомились, такие как очереди, служба хранения таблиц и служба SQL Azure. Этот

пример наглядно демонстрирует один из подходов к реализации приложения, поддерживающего горизонтальную масштабируемость.

Создание веб-роли

В этом примере мы создадим одну веб-роль, имеющую простой пользовательский интерфейс, включающий поле ввода, позволяющее указать количество комплектов игры и кнопку с надписью **Place Orders** (Заказать). Для реализации интерфейса мы будем использовать функциональность библиотеки jQuery Mobile в сочетании с шаблоном F#/C# ASP.NET Web API. Когда пользователь щелкнет на кнопке отправки заказа, система поместит сообщения в тему с именем `Orders`. Внешний вид формы заказа представлен на рис. 3.3. Это во многом искусственный пример, потому что настоящая форма заказа должна была бы включать массу дополнительной информации, такой как имя заказчика, адрес доставки заказа и способ оплаты. Однако он дает достаточно информации, чтобы на его основе можно было создать более сложное решение.

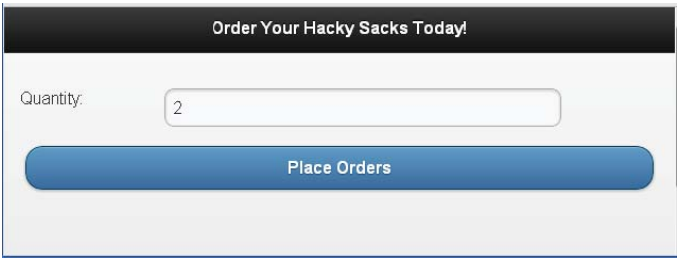

Рис. 3.3. Форма заказа комплектов игры в мешочек

Для этого я сначала создал решение F#/C# ASP.NET MVC 4 Web API, как описывается в разделе «Создание и развертывание приложений F# на платформе Azure», выше. Это веб-приложение – единственный издатель сообщений, которые будут получать и обрабатывать рабочие роли. Публикация сообщений будет выполняться с помощью библиотеки Fog, поэтому необходимо также установить NuGet-пакет `Fog`. Для имитации загрузки веб-роль будет передавать в очередь по одному сообщению для каждого заказанного комплекта игры.

Затем я добавил некоторый код на JavaScript. Хотя этого кода совсем немного, тем не менее, я поступил так, как поступил бы в действующем приложении. В частности, я использовал загрузчик

модулей RequireJS и подключил модуль, содержащий необходимый код на JavaScript, с его помощью. Так как основное внимание в этой книге уделяется языку F#, я не буду приводить этот код. Однако вы можете найти его по адресу: http://bit.ly/hacky-sack.

Код на JavaScript производит POST-запрос к соответствующей HTTP-службе, предоставляемой решением ASP.NET Web API. Подобную функциональность с той же легкостью можно было бы реализовать в стандартном приложении ASP.NET MVC, однако я предпочитаю определять HTTP-службы. Это упрощает организацию взаимодействий с другими приложениями. Кроме того, если позднее пришлось бы развертывать код, выполняемый на стороне клиента, как приложение на «родном» языке с помощью обертки, такой как PhoneGap, использование ASP.NET Web API помогло бы выполнить такой переход.

На контроллер ASP.NET Web API возлагается единственная обязанность – публикация сообщения с информацией о заказе в теме Orders. Для этого необходимо добавить параметры настройки поддержки Azure Service Bus в библиотеке Fog, изменить имя ValuesController на OrdersController и добавить код, как показано ниже:

```
namespace FsWeb.Controllers

open System
open System.Web.Http
open FsWeb.Models
open FsWeb.Commands

type OrdersController() =
     inherit ApiController()
     member x.Post (order:HackySackOrder) =
          [1..order.Quantity]
          |> Seq.iter(
               fun i ->
                    { RowKey = Guid.NewGuid().ToString()
                      PartitionKey = "Orders"
                      PlacementDateTime = DateTime.Now
                      Quantity = order.Quantity
                      Instance = i }
                    |> Fog.ServiceBus.Publish "Orders" )
```

Определение класса HackySackOrder, экземпляр которого ожидает получить метод Post, приводится ниже:

```
type HackySackOrder() =
     let mutable quantity = 0
     member x.Quantity with get() = quantity and set v = quantity <- v
```

PlaceOrderCommand

Возможно, вы обратили внимание, что в методе `Post` контроллера `OrdersController` я создаю некоторую запись, которая затем публикуется в теме. Это – запись с именем `PlaceOrderCommand`. Рабочие роли должны иметь возможность преобразовать сообщения, извлекаемые из темы, в записи этого типа, поэтому я поместил определение типа `PlaceOrderCommand` в новый проект с именем `Commands`. Экземпляры `PlaceOrderCommand` будут сохраняться и в службе хранения таблиц, и в службе SQL Azure. Как упоминалось в разделе «Таблицы», выше, служба хранения таблиц требует наличия в записях некоторых свойств со специальными именами, поэтому я добавил в тип `PlaceOrderCommand` свойства `RowKey` и `PartitionKey`. Это – пример, как можно использовать записи F# с хранилищем таблиц, вместо подхода на основе классов, показанного в разделе «Таблицы» выше. Определение `PlaceOrderCommand` показано ниже:

```
namespace FsWeb.Commands

open System
open System.ServiceModel
open System.Runtime.Serialization
open System.Data.Services.Common
open System.ComponentModel.DataAnnotations

[<DataContract>]
[<DataServiceKey("PartitionKey", "RowKey")>]
type PlaceOrderCommand =
     { [<DataMember>] mutable PartitionKey : string
       [<DataMember>] [<Key>] mutable RowKey : string
       [<DataMember>] mutable PlacementDateTime : DateTime
       [<DataMember>] mutable Quantity : int
       [<DataMember>] mutable Instance : int }
```

> **Примечание.** Как я уже упоминал несколько раз, версия F# 3.0 поддерживает атрибут `CLIMutable`, применение которого упрощает этот пример, позволяя не использовать ключевое слово `mutable` в записи `PlaceOrderCommand`. Примеры и дополнительные пояснения, касающиеся атрибута `CLIMutable`, я дам в главе 4.

Рабочие роли

Рабочие роли отвечают за «фактическое» размещение заказа, сохраняя его в службах SQL Azure и хранения таблиц. Чтобы продемонстрировать различные механизмы масштабирования, поддерживаемые платформой Azure, мы создадим один экземпляр рабочей роли, взаимодействующий со службой SQL, и два экземпляра, взаимодействующие со службой хранения таблиц. Две рабочие роли, добавляющие заказы в хранилище таблиц, будут конкурировать между собой за право извлечь сообщение, ожидающее обработки (этот прием известен как шаблон конкурирующих потребителей (competing consumer pattern)).

Для этого я добавил две рабочие роли в решение, созданное в разделе «Создание веб-роли», выше, одну с именем `OrderProcessor2` и одну с именем `OrderSQL`. Так как мне требуется создать два экземпляра рабочей роли `OrderProcessor2`, я открыл файл *Service Configuration.*.cscfg* (находится в проекте Windows Azure решения) и изменил количество экземпляров с 1 на 2. Наконец, я установил в новые проекты ролей NuGet-пакет `Fog`.

> **Примечание.** Файл ServiceConfiguration.*.cscfg будет содержать элементы для всех проектов в решении, поэтому будьте внимательны, измените значение счетчика экземпляров только в проекте `OrderProcessor2`.

Теперь, когда все проекты настроены, можно добавить код, реализующий взаимодействия со службой хранения таблиц. Как я уже упоминал, за это будет отвечать рабочая роль, обрабатывающая заказ (с именем `OrderProcessor2`). Разумеется, в действующую наверняка придется включить дополнительную логику, например, для обработки кредитной карты, рассылки извещений по электронной почте и так далее, но механизм взаимодействий с хранилищем останется практически без изменений. Сначала я добавил в файл *ServiceConfiguration.Local.cscfg* проекта Azure настройку доступа к службе хранения таблиц, необходимую для библиотеки `Fog`, как показано ниже:

```
<Setting name="TableStorageConnectionString"
     value="UseDevelopmentStorage=true" />
```

Ниже представлен класс `WorkerRole` роли `OrderProcessor2`, реализующий подписку на тему `Orders` и сохраняющий результат в хранилище таблиц:

```
type WorkerRole() =
     inherit RoleEntryPoint()

     let log message kind = Trace.WriteLine(message, kind)

     override wr.Run() =
          log "OrderProcessor2 entry point called" "Information"
          try
               Fog.ServiceBus.Subscribe "Orders" "OrderProcessor"
                    <| fun m ->
                         let entity = m.GetBody<PlaceOrderCommand>()
                         Fog.Storage.Table.CreateEntity "HackySackOrders" entity
                    <| fun ex m -> log ex.Message "Error"
          with
          | ex ->
               log ex.Message "Error"

          while(true) do
               Thread.Sleep(10000)

override wr.OnStart() =
     // Максимальное число конкурирующих соединений
     ServicePointManager.DefaultConnectionLimit <- 12
     // Информацию об особенностях обработки изменений в настройках
     // см. в статье на MSDN: http://go.microsoft.com/fwlink/?LinkId=166357.
     base.OnStart()
```

Код получился достаточно простым, потому что поступающие сообщения уже подготовлены для сохранения в таблице. Все, что остается здесь сделать – оформить подписку на тему `Orders` с именем `OrderProcessor` и предоставить функции обработки успешных и неудачных попыток обработки. В случае успеха тело сообщения сохраняется в таблице.

> **Внимание.** Цикл `while`, выделенный жирным в примере выше, играет очень важную роль. Без него реализация рабочей роли выполнила бы подписку и прекратила работу, из-за чего она не смогла бы принимать сообщения.

Рабочая роль SQL Azure

Теперь, когда у нас есть два экземпляра рабочей роли, взаимодействующие со службой хранения таблиц, настало время добавить реализацию рабочей роли для работы со службой SQL Azure

(с именем OrderSQLStore). Для организации взаимодействий со службой OrderSQLStore применяется фреймворк Entity Framework 5, использующий подход «сначала код» (code-first). С учетом имеющейся настройки учетной записи в SQL Azure и других конфигурационных параметров, код выглядит как комбинация примера из главы 1 с примером оформления подписки OrdersProcessor2. Код в примере ниже в основном связан с использованием фреймворка Entity Framework. Единственное существенное отличие от примера в главе 1 состоит в том, как извлекается строка подключения и передается экземпляру DbContext. В примере из главы 1 мы указывали имя строки подключения, которая затем извлекалась из конфигурационного файла. При работе с платформой Azure у нас нет такой возможности, поэтому для извлечения строки и передачи ее экземпляру DbContext, я добавил стандартный параметр настройки и использовал метод RoleEnvironment.GetConfigurationSettingValue:

```
namespace FsWeb.Repositories

open Microsoft.WindowsAzure.ServiceRuntime
open System.Data.Entity
open FsWeb.Commands

type HackySackStoreEntities() =
     inherit DbContext(
          RoleEnvironment.GetConfigurationSettingValue
          <| "HackySackStoreConnectionString")

     do Database.SetInitializer(
          CreateDatabaseIfNotExists<HackySackStoreEntities>())

     [<DefaultValue(true)>] val mutable
          orderCommands : DbSet<PlaceOrderCommand>
     member x.OrderCommands
          with get() = x.orderCommands and set v = x.orderCommands <- v
```

Код обработки сообщений оформляет подписку с именем OrderSQLStore и затем передает все сообщения функции, которая сохранит их в базе данных SQL Azure:

```
let handleNewOrderCommand entity =
     try
          use context = new HackySackStoreEntities()
```

```
            context.OrderCommands.Add entity |> ignore
            context.SaveChanges() |> ignore
        with
        | ex ->
            log ex.Message "Error"
            raise ex

override wr.Run() =
    log "OrderSQLStore entry point called" "Information"
    try
        Fog.ServiceBus.Subscribe "Orders" "OrderSQLStore"
            <| fun m ->
                m.GetBody<PlaceOrderCommand>()
                |> handleNewOrderCommand
            <| fun ex m -> log ex.Message "Error"
    with
    | ex -> log ex.Message "Error"

    while(true) do
        Thread.Sleep(10000)
```

Последние штрихи

Запустив все эти роли, вы обнаружите, что записи создаются сразу в двух хранилищах, в SQL Azure и в хранилище таблиц. Вы можете также заметить, что значок, обозначающий занятость сайта, отображается несколько дольше, чем можно было бы ожидать. Это обусловлено тем, что функция `Publish` выполняет свои операции синхронно. Так как наша цель – создать приложение с высокой отзывчивостью, такое его поведение просто недопустимо. С текущим синхронным подходом время отклика на `POST`-запрос с единственным пользователем будет увеличиваться с ростом количества заказов.

Решить эту проблему можно несколькими способами и, как обычно, F# делает это решение по-настоящему простым. Для этого достаточно заключить вызов `Publish` в блок `async` и запустить его. Измененный код, где новые строки выделены жирным, приводится ниже:

```
type OrdersController() =
    inherit ApiController()
    member x.Post (order:HackySackOrder) =
        [1..order.Quantity]
        |> Seq.iter(
            fun i ->
```

```
async {
     { RowKey = Guid.NewGuid().ToString()
       PartitionKey = "Orders"
       PlacementDateTime = DateTime.Now
       Quantity = order.Quantity
       Instance = i }
     |> Fog.ServiceBus.Publish "Orders"
} |> Async.Start )
```

На этом наш пример можно считать законченным. Однако есть еще несколько важных моментов, которых я хотел бы коснуться, способных помочь вам в создании масштабируемых решений.

Примечание. Примеры, что приводятся далее в книге, не являются улучшенными версиями данного примера, поэтому их следует рассматривать независимо.

Кеширование

Наряду с применением асинхронных механизмов, поддержки горизонтального масштабирования, распределения нагрузки и приемов функционального программирования, таких как мемоизация, кеширование является важной частью любого масштабируемого решения. Кеширование легко можно реализовать внутри процесса, однако при этом теряется некоторая его эффективность из-за ограничений на размер кеша, отсутствия возможности совместного доступа и потери кеша в случае перезапуска процесса. Эта проблема часто решается перемещением кеша за пределы процесса, в базу данных SQL, но в этом случае возникает проблема потери производительности.

Одно из решений, обеспечивающих неплохие результаты, заключается в использовании механизма распределенного кеширования. Это решение позволяет ослабить ограничения на размер кеша, устраняет связь между существованием кеша и жизненным циклом приложения, и обеспечивает доступность кеша всем приложениям, которые в нем нуждаются. Кроме того, несмотря на накладные расходы, связанные с помещением данных в кеш и их извлечением из кеша, общая производительность обычно оказывается существенно выше, чем в случае хранения кеша в базе данных SQL. Многие современные реализации распределенного кеширования также предлагают возможность локального кеширования, позволяющую еще больше снизить накладные расходы.

Платформа Windows Azure предоставляет поддержку распределенного кеширования в виде службы с именем... барабанная дробь... Windows Azure Caching. Инструкции по настройке и использованию этой службы можно найти по адресу: https://www.windowsazure.com/en-us/develop/net/how-to-guides/cache/. Библиотека Fog предоставляет обертку вокруг Azure Caching API в виде модуля `Fog.Caching`. Теперь вы сможете создать приложение Azure, либо с веб-ролью, либо с рабочей ролью, установить библиотеку Fog и использовать следующий код:

```
let rowKey = Guid.NewGuid().ToString()

{ RowKey = rowKey
  PartitionKey = "Orders"
  PlacementDateTime = DateTime.Now
  Quantity = order.Quantity
  Instance = 1 }
|> Put rowKey |> ignore
```

Выделенный фрагмент инициализирует запись `PlaceOrderCommand` (которая была показана в примере с заказами на комплекты игры в мешочек выше), используя выражение записи, и затем помещает ее в распределенный кеш Azure. Позднее это значение можно извлечь из кеша, как показано ниже:

```
let result = Get<PlaceOrderCommand> rowKey
```

Функция `Get` возвращает значение типа `Option`, что позволяет легко определить, существует ли значение в кеше, не волнуясь о ситуации разыменования пустой ссылки.

> **Примечание.** Во всех примерах проектов на платформе Azure, приводившихся до сих пор, настройки сохранялись в файлах *.cscfg. Кеширование несколько отличается в этом смысле – все необходимые настройки должны сохраняться в конфигурационном файле приложения или веб-службы.

CDN и автоматическое масштабирование

Платформа Windows Azure имеет еще две особенности, которые я хотел бы осветить в этом разделе. Несмотря на то, что язык F# не предоставляет каких-либо преимуществ при использовании этих двух особенностей, те не менее, они могут сыграть важную роль в

создании масштабируемых приложений на F#, выполняющихся под управлением Windows Azure. Первая из них – поддержка *сети доставки содержимого* (Content Delivery Network, CDN) и вторая – *автоматическое масштабирование* (autoscaling).

Сети доставки содержимого являются еще одним важным винтиком в механизме масштабируемых приложений. Платформа Windows Azure позволяет использовать CDN для распространения кешированных двоичных объектов (blobs), хранящихся в службе хранения больших двоичных объектов. Это не только позволяет кешировать содержимое, но и приближает содержимое к пользователю. Чем ближе к пользователю окажется содержимое, тем быстрее оно будет доставлено.

Чтобы реализовать на языке F# передачу данных в Windows Azure CDN, достаточно с помощью библиотеки Fog сохранить их в хранилище больших двоичных объектов. После этого можно будет настроить новую конечную точку CDN, следуя инструкциям на вебстранице Windows Azure CDN http://bit.ly/using-cdn.

С помощью приемов, обсуждавшихся выше, мы легко сможем масштабировать свои решения, добавляя дополнительные серверы, действующие совместно для достижения поставленной цели. Часто приложения, построенные на основе этого шаблона, должны учитывать вероятность превышения расчетной нагрузки и иметь некоторый запас на непредвиденные обстоятельства. Дешевле купить дополнительные аппаратные средства, чем терять доходы на продажах из-за потери доверия со стороны клиентов, когда система оказывается не в состоянии обслужить их в случае неожиданно высокого наплыва посетителей.

В идеале можно развернуть решение в конфигурации для расчетной нагрузки, а затем увеличивать или уменьшать вычислительные мощности, опираясь на статистику использования или знания о периодах пиковых нагрузок. Именно такую возможность дает механизм автоматического масштабирования. С помощью нескольких параметров настройки и строк кода можно определить правила, которые позволят вашему решению изменять свой масштаб в соответствии с требованиями. Необходимые инструкции по настройке можно найти по адресу: http://bit.ly/using-cdn.

Блистательные примеры на F#

Команда разработчиков F# вместе с сообществом пользователей F# реализовали несколько превосходных библиотек и фреймворков для взаимодействия с платформой Windows Azure. В этом разделе я

познакомлю вас с некоторыми, наиболее яркими примерами. Перечисленные ниже библиотеки и фреймворки используют всю мощь языка F# для своих задач, решить которые было бы намного труднее на таких языках, как C# или VB.NET.

Помимо представленных здесь библиотек и фреймворков, на просторах Интернета можно найти множество видеороликов и статей, содержащих массу интереснейшей информации и примеров использования платформы Azure из программ на F#. Например, Дон Сайм (Don Syme) (создатель языка F#) в своем блоге (http://bit.ly/don-syme) рассказывает о возможностях использования Windows Azure из F#. Как еще один пример, Ноа Гифт (Noah Gift) опубликовал превосходную статью (http://bit.ly/parsing-logfiles) с описанием применения F# и Azure для парсинга файлов журналов.

{m}brace

Возможности языка F# особенно ярко проявляются в области распределенных вычислений. Такие его особенности, как вычислительные выражения и класс `MailboxProcessor` (он же агент), открывают путь к вычислениям, основанным на конкурентной передаче сообщений. Но, несмотря на большое количество преимуществ, многое еще приходится делать самостоятельно.

Компания Nessos Information Technologies создала фреймворк и библиотеку времени выполнения, существенно упрощающие создание решений распределенных вычислений на F#. Подобно языку Erlang, обладающему многими уникальными особенностями, фреймворк и среда выполнения (с названием {m}brace) предоставляют прозрачную поддержку распределенных вычислений и похвально большое количество дополнительных возможностей, включая «горячую» замену кода, его оптимизацию, средства управления потоками данных и их обработкой, обширное журналирование и трассировка, и комбинаторы акторов.

В качестве примера одной из множества особенностей, рассмотрим следующий фрагмент, который использует малую часть фреймворка акторов, входящего в состав {m}brace:

```
type PingPong = | Ping
let rec pingPongBehavior (self: Actor<PingPong>) = async {
    let! message = self.Receive()

    match message with
```

```
    | Ping ->
        log
        <| sprintf "Pong received at %O by %s" DateTime.Now self.Name
        <| "Information"

    return! pingPongBehavior self }

let actor1 = Actor.bind "actor1" pingPongBehavior
let actor2 = Actor.bind "actor2" pingPongBehavior
let supervisor = Actor.broadcast [actor1; actor2]

supervisor.Start()

!supervisor <-- Ping
```

Первая функция, `pingPongBehavior`, мало чем отличается от примера использования `MailboxProcessor` в главе 1. Все самое интересное начинается с определений `actor1`, `actor2` и `supervisor`. `actor1` и `actor2` — это всего лишь привязки имен к экземплярам акторов, где функция `pingPongBehavior` определяет логику поведения каждого экземпляра актора. Далее выполняется настройка актора `supervisor`, посылающего сообщения всем другим акторам. За кулисами функция `broadcast` выполняет связывание акторов `actor1` и `actor2` с актором `superviser`. Эта позволяет актору `supervisor` не только рассылать сообщения связанным с ним акторам, но так же контролировать запуск и остановку этих акторов.

Несколько слайдов и множество отличных примеров можно найти по адресу: http://bit.ly/m-brace. Кроме того, имеется видеозапись презентации по адресу: http://bit.ly/skills-matter.

Cloud Numerics

Библиотека Cloud Numerics обеспечивает поддержку масштабируемости приложений, реализующих числовую обработку и анализ больших объемов данных, для выполнения которых требуются значительные вычислительные мощности. Руководство по настройке, а также несколько примеров можно найти по адресу: http://social.technet.microsoft.com/wiki/contents/articles/5993.microsoft-codename-cloud-numerics.aspx. Дополнительную информацию можно также найти в блогах MSDN (http://bit.ly/cloud-numerics).

Библиотека обладает несколькими особенностями, включая методы обработки массивов числовой информации и определения

структур данных, которые с успехом могут использоваться в распределенном окружении, шаблон проекта и пример приложения, а также утилиту развертывания, упрощающую настройку и публикацию приложений в кластере Azure.

Hadoop MapReduce для .NET

Hadoop – это фреймворк, хорошо известный среди тех, кто занимается обработкой больших объемов данных. Он поддерживает распределенную обработку данных с применением кластеров огромных размеров. В настоящее время платформа Windows Azure уже включает поддержку служб Hadoop. На момент написания этих строк, поддержка служб Hadoop в Azure находилась на стадии реализации и была доступна только по приглашениям. Запросить приглашение можно по адресу: https://www.hadooponazure.com.

В текущей реализации лучшим способом взаимодействия с Hadoop из F# является применение фреймворка Hadoop Streaming. Карл Нолан (Carl Nolan) представил несколько примеров на языке F# использования великолепной библиотеки[1], ознакомившись с которыми вы легко сможете реализовать собственные задания для Hadoop Streaming MapReduce. Кроме того, Карл написал несколько статей, где можно найти дополнительную информацию и примеры[2].

В заключение

F# и Windows Azure прекрасно уживаются друг с другом. Как я показывал на протяжении всей главы, взаимодействие с платформой Windows Azure через Windows Azure SDK для .NET из F# реализуется даже проще, чем из C# или VB.NET, а библиотека Fog еще больше упрощает дело. Кроме того, такие фреймворки как {m}brace, Cloud Numerics, Hadoop Streaming и F# MapReduce позволяют использовать всю мощь языка F# для обработки огромных объемов данных и/или массивных вычислений.

В следующей главе мы рассмотрим дополнительные способы увеличения скорости работы веб-решений на языке F# и приемы их масштабирования до огромных пропорций. Я покажу, как создавать вебсокеты в мобильных и веб-приложениях с помощью SignalR, как сохранять данные с помощью различных NoSQL-решений, и многое другое.

---

[1] http://bit.ly/hadoop-streaming.

[2] http://blogs.msdn.com/b/carlnol/.

Глава 4. Создание масштабируемых мобильных и веб-приложений

Главное в науке не столько получение новых фактов, сколько поиск новых способов их интерпретации.

– Сэр Уильям Брэгг (Sir William Bragg)

Возможно вам, как и мне, требуется не так много, чтобы вспомнить свой первый опыт общения с компьютером. Фактически, к концу предыдущего предложения вы наверняка мысленно перенеслись в прошлое и вспомнили те ощущения.

Впервые я прикоснулся к компьютеру в 1985, когда мои родители принесли домой IBM PC. Он был украшен двумя приводами 5- и 7-дюймовых дискет и имел замечательный зеленый монитор. Я сразу же был заинтригован новой игрушкой и быстро погрузился в чтение различных инструкций, поставлявшихся вместе с таинственной машиной, чтобы изучить и использовать секреты волшебства. Несколько успешных строк на BASICA, и я был покорен.

С тех пор в компьютерном мире произошли гигантские изменения, но многие из основных понятий остались прежними. В процессе этих изменений часто открывались новые способы интерпретации существующих фактов. И без этих новых способов мы наверняка не были бы там, где находимся сейчас.

Каждый день появляются новые способы интерпретации старых фактов и непрекращающееся движение вперед яркое тому подтверждение. Один из положительных побочных эффектов этого движения выражается в том, что все больше и больше людей получают доступ к сети Интернет. Эти пользователи Всемирной Паутины ожидают, что сайты будут отзывчивыми и их можно будет просматривать на самых разных современных устройствах. Другим положительным

эффектом стала возможность подключения к Интернету не только с традиционных, стационарных компьютеров, но и с мобильных устройств. В статье на сайте CNET News[1] отмечается, что по заявлению ассоциации CTIA Wireless Association количество подключений к беспроводным сетям на территории США превышает численность населения.

Чтобы обеспечить отзывчивость и адаптивность решений, масштабирующихся соответственно числу пользователей, мы так же должны открывать новые способы интерпретации различных технологий и подходов. Я уже показал несколько способов, которые помогут увеличить скорость и масштабируемость ваших решений. И в этой главе я познакомлю вас с дополнительными приемами и примерами на F# в этом же направлении.

Масштабирование с применением веб-сокетов

Веб-сокеты (Web Sockets) появились совсем недавно. Эта широко разрекламированная особенность является частью развивающейся спецификации HTML5. Хотя эта спецификация продолжает изменяться, это не мешает основным игрокам на рынке встраивать поддержку HTML5 в свои браузеры. Особенность, получившая название веб-сокеты (Web Sockets) – не исключение.

Веб-сокеты предоставляют возможность двустороннего обмена информацией между клиентом и сервером. Они могут сделать ваш сайт более быстрым и более масштабируемым за счет снижения количества запросов и уменьшения объема передаваемой информации. Кроме того, наличие канала полноценной двусторонней связи позволяет обновлять страницы почти мгновенно. Дополнительную информацию о преимуществах веб-сокетов можно найти по адресу: http://www.websocket.org/quantum.html[2].

> **Примечание.** Несмотря на то, что веб-сокеты способны увеличить производительность и масштабируемость вашего сайта, все же не следует злоупотреблять ими. Дополнительные сведения по этой теме можно найти по адресу: http://bit.ly/html5-sockets.

---

[1] http://cnet.co/XdKGBm.

[2] Для русскоязычных читателей можно порекомендовать статью http://msdn.microsoft.com/ru-ru/magazine/hh975342.aspx. – *Прим. перев.*

Возможно, вам интересно узнать, в каких типах решений или в каких областях внутри решений можно извлечь максимальную выгоду от применения веб-сокетов. Типичный ответ на этот вопрос: «везде, где требуется обновлять информацию в реальном масштабе времени». Но такой ответ выглядит слишком обобщенным. Вот несколько более конкретных идей: сайт, поставляющий статистическую информацию и отчеты в реальном масштабе времени, отображение текущих котировок на бирже, каноническим примером может служить обмен сообщениями в реальном времени, игры, производственные системы, а также отправка пользователю извещений и предупреждений. На практике, вам нужно самим оценить, где веб-сокеты дают наибольшие выгоды, и использовать их только в этих областях.

Пример использования веб-сокетов на платформе .NET 4.5 и IIS 8

На выбор есть несколько возможностей создания серверов веб-сокетов. Версии IIS до IIS 8 не поддерживают веб-сокеты. Поэтому, если вам необходима поддержка веб-сокетов в IIS, используйте версию IIS 8 и .NET 4.5. Пол Батум (Paul Batum) в своем блоге опубликовал замечательную статью (http://www.paulbatum.com/2011/09/getting-started-with-websockets-in.html), где подробно рассказывает об этом и демонстрирует пример на C#. В примере на F#, который следует ниже, используется похожий прием.

> **Примечание.** Существует также неподдерживаемый прототип реализации веб-сокетов, которая теоретически должна работать в IIS более ранних версий. Найти ее можно по адресу: http://bit.ly/sockets-prototype.

Давайте создадим простое приложение опроса посетителей, которое позволит просматривать результаты в реальном времени с помощью веб-сокетов. В нашем опросе будет всего один вопрос: «Какой язык программирования вы предпочитаете?». И возможные варианты ответов: F#, C#, Erlang, JavaScript и «Другой». Благодаря применению шаблонов Twitter Bootstrap CSS, веб-версия формы опроса будет выглядеть, как показано на рис. 4.1. На рисунке изображено два открытых окна браузера Chrome, в каждом из которых загружена форма опроса. Это позволит нам оценить отзывчивость веб-сайта, использующего веб-сокеты.

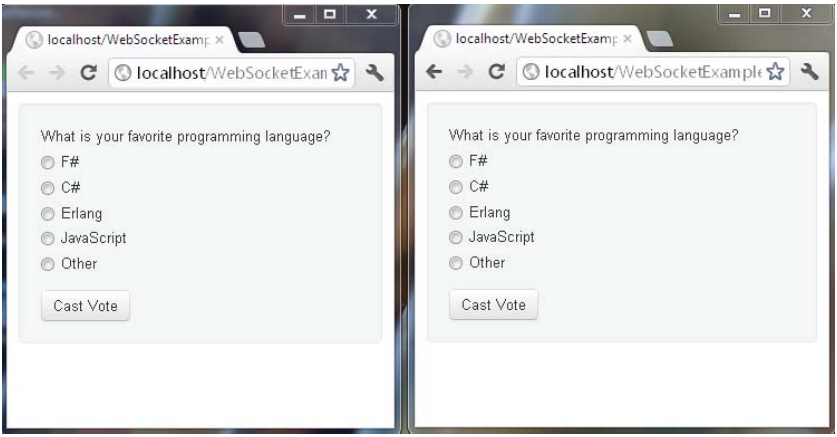

Рис. 4.1. Форма опроса

Примечание. Чтобы опробовать этот пример, вам потребуется запустить Visual Studio в режиме администратора.

Представление, которое выводится после ответа на вопрос, отображает диаграмму, отражающую результаты опроса. Это представление будет обновляться в реальном масштабе времени. Создаваться диаграмма будет с помощью JavaScript-библиотеки D3[1]. Но, поскольку это не связано непосредственно с веб-сокетами, я опущу код, использующий функции из библиотеки D3. Но вы сможете найти его в файле `mainModule.js`, в проекте WebSocketIIS8Example[2].

Создайте новое приложение ASP.NET MVC 4 и убедитесь, что все проекты используют .NET 4.5. Теперь можно установить NuGet-пакет `Microsoft.WebSockets`. Можно также добавить новый файл *.ashx* в проект на C# и несколько файлов поддержки *.fs* в проект F# WebApp.

Первый файл *.fs* в этом примере имеет имя *ChartWebSocket.ashx.fs*. Концептуально он напоминает файл *.cs*, который часто создается при добавлении элемента *.ashx* в проект на C#. Код в этом файле определяет класс, реализующий интерфейс `IHttpHandler`. Когда в вебсокет поступает новый запрос, создается новый экземпляр класса `WebSocketChartHandler`, как показано ниже:

---

[1] http://d3js.org/.

[2] https://github.com/dmohl/fs-web-cloud-mobile/tree/master/Ch%204.

```
namespace FsWeb

open System
open System.Web
open Microsoft.Web.WebSockets
open WebSocketServer

type ChartWebSocket() =
     interface IHttpHandler with
          member x.ProcessRequest context =
               if context.IsWebSocketRequest then
                    context.AcceptWebSocketRequest(new WebSocketChartHandler())
          member x.IsReusable = true
```

Следующий файл, *WebSocketChartHandler.fs*, делится на несколько частей. В первой части создается экземпляр `WebSocketCollection` для хранения информации о всех подключенных клиентах. Во второй части определяется запись `VoteCounts`. Она будет использоваться как контейнер для последующей сериализации в формат JSON и передачи каждому клиенту через веб-сокеты. Эти две части кода выглядят, как показано ниже:

```
let mutable clients = WebSocketCollection()

type VoteCounts = { language : string; count : int }
```

Для хранения количества голосов, отданных за каждый язык программирования, используется экземпляр `MailboxProcessor`. Этот пример легко можно было бы переделать для сохранения результатов опроса в базе данных, но ради простоты мы будем хранить их в памяти. Реализация `MailboxProcessor` показана ниже:

```
type Message =
     | Vote of string * AsyncReplyChannel<seq<string*int>>

let votesAgent = MailboxProcessor.Start(fun inbox ->
     let rec loop votes =
          async {
               let! message = inbox.Receive()
               match message with
               | Vote(language, replyChannel) ->
                    let newVotes = language::votes
                    newVotes
```

```
                |> Seq.countBy(fun lang -> lang)
                |> replyChannel.Reply
                do! loop(newVotes)
            do! loop votes
        }
    loop List.empty)
```

Данная реализация `MailboxProcessor` мало чем отличается от той, что была представлена в главе 1, поэтому я остановлюсь лишь на выделенном коде.

Когда из виртуальной очереди `Vote` извлекается очередное сообщение, оно добавляется в список. Списки в языке F# – это *односвязные списки* структур данных, показывающие высокую производительность, оставаясь при этом неизменными. Высокая производительность списков обусловлена тем, что копия списка не создается заново при добавлении нового элемента. Вместо этого создается новый элемент и в него добавляется ссылка на прежний список. Для этого мы используем оператор *cons* (`::`). После завершения операции, новый результат опроса оказывается в начале списка.

Далее выполняется подсчет голосов для каждого языка и возвращается последовательность `string * int`. Например, если за F# было отдано три голоса, за C# – два и один за JavaScript, в экземпляр `MailboxProcessor` будет помещен результат в виде последовательности, содержащей: `F#, 4; C#, 1; JavaScript, 2`.

Последняя часть в файле *WebSocketChartHandler.fs* – определение класса `WebSocketChartHandler`. Этот класс наследует `WebSocketHandler` и переопределяет три метода родителя, как показано ниже:

```
type WebSocketChartHandler() =
    inherit WebSocketHandler()

    override x.OnOpen() = clients.Add x
    override x.OnMessage(language:string) =
        votesAgent.PostAndReply(fun reply -> Message.Vote(language, reply))
        |> Seq.map(fun v -> { language = fst v; count = snd v } )
        |> JsonConvert.SerializeObject
        |> clients.Broadcast
    override x.OnClose() =
        clients.Remove x |> ignore
```

Имена методов говорят сами за себя, поэтому я остановлюсь только на выделенном коде в методе `OnMessage`. Первая строка отсылает

голос экземпляру MailboxProcessor и ожидает ответа. Результат этого вызова передается функции Seq.map для отображения результатов в последовательность записей VoteCounts, описанных выше. Затем эта последовательность преобразуется в формат JSON с помощью Json.NET. И, наконец, результат рассылается всем клиентам через веб-сокеты.

Ниже приводится сокращенная версия сценария на JavaScript, который выполняет подключение к серверу веб-сокетов:

```
$(function () {
    var uri,
        updateChart,
        $pages = $(".page");

    $pages.hide();
    $pages.first().toggle();

    updateChart = function (data) {
        /* Код удален для экономии места */
    };

    uri = "ws://localhost/WebSocketExample/ChartWebSocket.ashx";

    websocket = new WebSocket(uri);

    websocket.onopen = function () {
        $("#vote").click(function (event) {
            var vote = $("input:radio[name=langOption]:checked");

            if (vote.length) {
                websocket.send(vote.val());
            };

            $pages.hide();
            $("#results").toggle();

            event.preventDefault();
        });
    };

    websocket.onmessage = function (event) {
        updateChart($.parseJSON(event.data));
    };
});
```

Увидеть пример в действии, можно, открыв два окна браузера, как показано на рис. 4.1, и проголосовав за любимый язык (или языки). В каждом браузере будет отображаться диаграмма с общим числом голосов, отданных за каждый язык. Кроме того, диаграмма в каждом браузере будет обновляться автоматически, и отображать самые последние результаты голосования почти немедленно, сразу после щелчка на кнопке **Cast Vote** (Голосовать) любым клиентом. Как это выглядит показано на рис. 4.2.

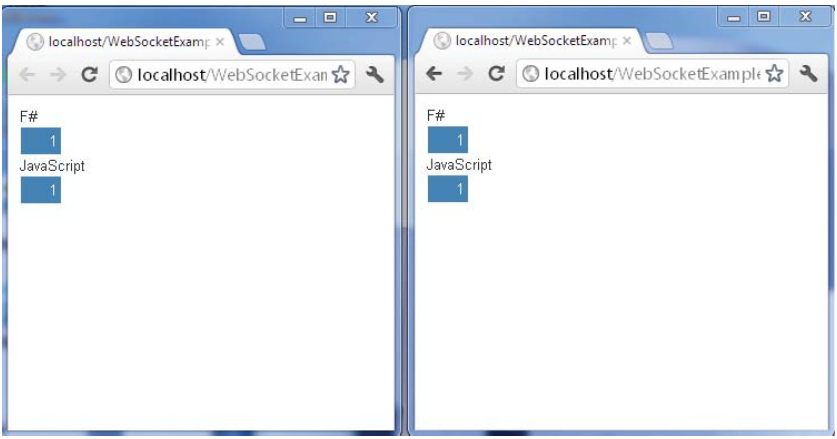

Рис. 4.2. Результаты опроса

Создание сервера веб-сокетов с помощью Fleck

Пример, который я только что представил, работает вполне прилично, но как быть, если в нашем распоряжении нет веб-сервера IIS 8, или если нет возможности обновиться до версии .NET 4.5, или если требуется использовать веб-сокеты за пределами IIS? К счастью в о всех этих случаях у нас на выбор есть несколько вариантов. Обычно я отдаю предпочтение библиотеке Fleck. Чтобы воспользоваться этой библиотекой, установите NuGet-пакет `Fleck`. После этого вы сможете использовать код, как показано ниже, реализующий автономный сервер веб-сокетов:

```
module FsFleckServer

open System
```

```
open System.Collections.Generic
open Fleck
open Newtonsoft.Json

type VoteCounts = { language : string; count : int }

type Message =
     | Vote of string * AsyncReplyChannel<seq<string*int>>

let votesAgent = MailboxProcessor.Start(fun inbox ->
     let rec loop votes =
          async {
               let! message = inbox.Receive()
               match message with
               | Vote(language, replyChannel) ->
                    let newVotes = language::votes
                    newVotes
                    |> Seq.countBy(fun lang -> lang)
                    |> replyChannel.Reply
                    do! loop(newVotes)
               do! loop votes
          }
     loop List.empty)

let main() =
     FleckLog.Level <- LogLevel.Debug
     let clients = List<IWebSocketConnection>()
     use server = new WebSocketServer "ws://localhost:8181"

     let socketOnOpen socket = clients.Add socket

     let socketOnClose socket = clients.Remove socket |> ignore

     let socketOnMessage language =
          let results =
               votesAgent.PostAndReply(fun reply -> Message.Vote(language, reply))
               |> Seq.map(fun v -> { language = fst v; count = snd v } )
               |> JsonConvert.SerializeObject
          clients |> Seq.iter(fun c -> c.Send results)

     server.Start(fun socket ->
                    socket.OnOpen <- fun () -> socketOnOpen socket
                    socket.OnClose <- fun () -> socketOnClose socket
                    socket.OnMessage <- fun message -> socketOnMessage message)
```

```
    Console.ReadLine() |> ignore

main()
```

О функции using

Я уже несколько раз говорил о ключевом слове `use` и упоминал функцию `using`, но я не показывал не примерах, как можно использовать эту функцию. Функция `using` позволяет получить более полный контроль над тем, когда будет вызываться `Dispose()`. Ниже показано, как можно реализовать модуль `main` для примера использования библиотеки Fleck с помощью функции `using`:

```
using (new WebSocketServer "ws://localhost:8181")
    (fun server ->
      let socketOnOpen socket = clients.Add socket

      let socketOnClose socket = clients.Remove socket |> ignore

      let socketOnMessage language =
          let results =
              votesAgent.PostAndReply(fun reply ->
                                      Message.Vote(language, reply))
              |> Seq.map(fun v -> { language = fst v; count = snd v } )
              |> JsonConvert.SerializeObject
          clients |> Seq.iter(fun c -> c.Send results)

      server.Start(fun socket ->
                   socket.OnOpen <- fun () -> socketOnOpen socket
                   socket.OnClose <- fun () -> socketOnClose socket
                   socket.OnMessage <- fun message ->
                                       socketOnMessage message)

      Console.ReadLine() |> ignore)
```

Определения двух типов и агента теми же, что и в примере с использованием IIS 8. Выделенный код создает экземпляр `WebSocketServer`; определяет три метода для обработки событий открытия сокета, закрытия сокета и получения сообщения, соответственно; и запускает сервер.

Чтобы опробовать этот пример, достаточно изменить значение `uri` в сценарии JavaScript, указав URL сервера Fleck, и очистить кеш

браузера. Это приложение будет действовать точно так же, как и пример на основе IIS 8.

SignalR

Я полагаю, все согласятся, если сказать, что веб-сокеты – это круто! Двунаправленный обмен данными, который они поддерживают, открывает массу новых возможностей. К сожалению, поддержка веб-сокетов пока не получила повсеместного распространения. Поэтому, несмотря на то, что за веб-сокетами большое будущее, они не выглядят достаточно жизнеспособным выбором для систем, которые нужно создавать уже сегодня. Если только у нас нет способа использовать преимущества веб-сокетов, когда они доступны, и задействовать другой подход в противном случае.

Думаю, вы уже догадались, что такие возможности существуют и одной из них является комплект библиотек под общим названием SignalR. Комплект SignalR начинал разрабатываться двумя сотрудниками корпорации Microsoft как открытый проект, который позднее стал официально поддерживаться корпорацией Microsoft. SignalR упрощает создание решений, действующих в масштабе реального времени и использующих прием обмена асинхронными сигналами. SignalR позволяет разрабатывать современные решения, способные автоматически использовать веб-сокеты, когда они доступны.

Итак, что же такого делают библиотеки SignalR, когда поддержка веб-сокетов недоступна? Они проверяют наличие нескольких различных вариантов взаимодействий и используют тот, что лучше подходит для текущей задачи. Сначала проверяется поддержка веб-сокетов. Если она отсутствует, следом проверяется поддержка механизма событий, посылаемых сервером (Server-Sent Events, SSE), возможность обмена с использованием невидимого фрейма `iframe` (Forever Frame) и возможность организации длительного HTTP-соединения (Long Polling).

Примечание. SignalR – не единственный выбор, когда требуется использовать иные приемы организации взаимодействий, такие как Forever Frame и Long Polling. В качестве примеров можно также назвать библиотеки Socket.IO и NowJS. Однако библиотеки SignalR проще в использовании и к тому же разрабатывались с прицелом на работу с ASP.NET.

Пример создания постоянного соединения

При использовании SignalR для создания серверной части приложения, доступны два основных варианта. Первый – организация постоянного соединения, и второй – хаб (hub). Постоянное соединение очень похоже на то, что мы видели в примере использования библиотеки Fleck и веб-сокетов IIS 8.

Как вы увидите далее, для создания постоянного соединения требуется не так много кода. Для начала установите NuGet-пакет `SignalR`. Затем добавьте карту маршрутов и реализуйте класс, наследующий `PersistentConnection`. В следующем примере показан фрагмент файла *Global.fs*, где новый код выделен жирным:

```
type Global() =
     inherit System.Web.HttpApplication()

     static member RegisterRoutes(routes:RouteCollection) =
          routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
          routes.MapRoute("Default",
                          "{controller}/{action}/{id}",
                          { controller = "Home"; action = "Index"
                            id = UrlParameter.Optional } )
     member this.Start() =
          RouteTable.Routes
               .MapConnection<ChartServer>("chartserver",
                                           "chartserver/{*operation}") |> ignore

          AreaRegistration.RegisterAllAreas()
          Global.RegisterRoutes(RouteTable.Routes)
```

Объявление нового класса, наследующего `PersistentConnection`, показано в следующем примере. Как уже упоминалось, он не сильно отличается от примеров, использующих веб-сокеты IIS 8. Здесь вместо метода `OnMessage` мы переопределяем метод `OnReceivedAsync`:

```
module SignalRExample

open System
open SignalR
open Newtonsoft.Json

// Код, не изменившийся по сравнению с предыдущими примерами,
// был удален для экономии места
```

```
type ChartServer() =
     inherit PersistentConnection()

     override x.OnReceivedAsync(request, connectionId, data) =
          votesAgent.PostAndReply(fun reply -> Message.Vote(data, reply))
          |> Seq.map(fun v -> { language = fst v; count = snd v } )
          |> JsonConvert.SerializeObject
          |> base.Connection.Broadcast
```

Клиент на JavaScript

Клиентская часть приложения на основе SignalR может отличаться, в зависимости от того, на каком языке она написана, JavaScript или F#, а также от выбранной технологии на стороне сервера, `PersistentConnection` или `Hub`. Кроме того, сценарий на JavaScript имеет отличия от сценария, представленного выше и использующего вебсокеты. Эта разница отчасти обусловлена необходимостью использовать расширение для библиотеки jQuery, помогающее библиотекам SignalR творить свое волшебство. Ниже приводится сценарий на JavaScript для случая использования `PersistentConnection`, реализованного в примере выше:

```
/* Некоторый код был удален для экономии места */

var connection = $.connection("/chartserver");

connection.received(function (data) {
     updateChart($.parseJSON(data));
});

$('#vote').click(function (event) {
     var vote = $('input:radio[name=langOption]:checked');

     if (vote.length) {
          connection.send(vote.val());
     };

     $pages.hide();
     $("#results").toggle();

     event.preventDefault();
});

connection.start();
```

Использование этого клиента и сервера дает тот же результат, что и примеры на основе веб-сокетов. Страница с результатами имеет тот же внешний вид, как показано на рис. 4.2.

Клиент на F#

А что если потребуется связать сервер SignalR с клиентом, написанным на F#, а не на JavaScript? Нет проблем! В следующем примере приводится реализация консольного приложения на F#, с помощью которого можно проголосовать за свой любимый язык программирования.

Чтобы получить его, создайте новое консольное приложение на F#, установите NuGet-пакет `SignalR.Client` и добавьте следующий код:

```
module SignalRExample

open System
open SignalR.Client

let connection =
    Connection "http://localhost:2920/chartserver"

connection.Start().Wait()

connection.Send "F#"
|> Async.AwaitIAsyncResult
|> Async.Ignore
|> ignore

connection.Stop() |> ignore

printfn "Vote cast for F#"
Console.ReadLine() |> ignore
```

Пример создания хаба

Как я уже упоминал выше, библиотеки SignalR поддерживают еще один способ организации взаимодействий – на основе класса `Hub`. Этот способ предоставляет более высокоуровневую абстракцию, опирающуюся на класс `PersistentConnection`. Данная абстракция дает серверу возможность вызывать функции JavaScript по их именам. Рассмотрим пример.

Серверная сторона

Ниже демонстрируется реализация автономного сервера SignalR на основе класса `Hub`. Это – консольное приложение на F#, использующее пакет `SignalR.Hosting.Self`. Чтобы настроить хаб, нужно создать класс, наследующий `Hub`. Затем добавить методы, которые должны быть доступны клиентам. Вызвать функцию на JavaScript можно с помощью динамического объекта `Clients`, указав за ним имя вызываемой функции. Этот код выделен в следующем примере:

```
// Код из предыдущего примера опущен для экономии места
type ChartHub() =
     inherit Hub()
     member x.Send (data:string) =
          let result =
              votesAgent.PostAndReply(fun reply -> Message.Vote(data, reply))
              |> Seq.map(fun v -> { language = fst v; count = snd v } )
          try
              base.Clients?updateChart(result)
          with
          | ex ->
              printfn "%s" ex.Message

let server = Server "http://*:8181/"
server.MapHubs() |> ignore

server.Start()

printfn "Now listening on port 8181"
Console.ReadLine() |> ignore
```

В предыдущем фрагменте можно заметить оператор `?`, применяемый к динамическому объекту `Clients`. В языке F# не поддерживается точная копия динамических объектов, имеющихся в языке C#. Но в нем имеется более мощная возможность, позволяющая реализовать любую необходимую динамическую функциональность. Эта возможность поддерживается в форме *динамических операторов поиска* (dynamic lookup operators).

Динамические операторы поиска способны обеспечить ту же функциональность, что предлагается в языке C#, но при этом они не ограничиваются только этой функциональностью. Вы можете адаптировать реализацию динамических операторов поиска под свои нужды, расширяя их возможности далеко за рамки, установленные

в родственном языке. Одна из реализаций, завоевавших большую популярность, доступна в в составе библиотеки `ImpromptuInterface.FSharp`, которую можно установить как NuGet-пакет с тем же именем. Эта библиотека используется в примере на основе класса `Hub`.

Клиентская сторона

Использование класса `Hub` влечет за собой необходимость изменить код на JavaScript. Ниже приводится обновленная версия сценария на JavaScript, где жирным выделен код, необходимый для взаимодействия с классом `Hub`:

```
$.connection.hub.url = 'http://localhost:8181/signalr'

var chartHub = $.connection.chartHub;

chartHub.updateChart = function (data) {
     updateChart(data);
};

$('#vote').click(function (event) {
     var vote = $('input:radio[name=langOption]:checked');

     if (vote.length) {
          chartHub.send(vote.val())
     }

     $pages.hide();
     $("#results").toggle();

     event.preventDefault();
});

$.connection.hub.start();
```

Теперь осталось сделать еще один важный шаг, чтобы обеспечить работоспособность этого примера – добавить несколько ссылок на сценарии:

```
<script
  src="@Url.Content("~/Scripts/jquery.signalR-0.5.2.min.js")"></script>
<script
  src="http://localhost:8181/signalr/hubs" type="text/javascript"></script>
```

Обретаем мобильность

Добавить поддержку мобильных устройств, взаимодействия с которыми также выполняются с помощью библиотек SignalR, совсем несложно, особенно теперь, когда у нас уже есть некоторый фундамент. В этом разделе я познакомлю вас с несколькими разными способами включение в простое приложение опроса поддержки мобильных устройств. Первый заключается в использовании библиотеки jQuery Mobile. Второй – в создании приложения для платформы Windows Phone 7.

Способ на основе jQuery Mobile

Библиотека jQuery Mobile уже использовалась выше в этой книге, и я упоминал, что проект ASP.NET MVC 4 легко можно настроить, чтобы он по-разному реагировал на разные устройства. Однако, я фактически не демонстрировал эти настройки на примерах. Так как у нас уже имеется веб-версия приложения для проведения опроса, это – отличная возможность быстро пройтись по всем необходимым настройкам и обеспечить переключение представлений в веб-приложении ASP.NET MVC 4, когда доступ к странице осуществляется с помощью мобильного устройства.

Для начала нужно добавить новый файл с именем *_Layout.mobile.cshtml*. Затем в него следует добавить обычные ссылки на библиотеку jQuery Mobile, а также ссылки на библиотеки SignalR, D3 и JavaScript-сценарий `mainModule`, которые выше включались в оригинальный файл *_Layout*.

Теперь осталось лишь внести несколько изменений в разметку HTML в уже имеющемся файле *Index.cshtml file*. Благодаря этим изменениям, оба представления смогут использовать один и тот же файл *Index.cshtml*. Разметка с изменениями, выделенными жирным, показана ниже:

```
<div id="survey" class="page" data-role="page" >
  <div class="row" data-role="content">
    <form class="well">
      <fieldset data-role="controlgroup">
        <legend>What is your favorite programming language?</legend>
        <div class="controls">
          <label class="radio">
            <input type="radio" name="langOption"
```

```
                id="fsharp" value="F#" />F#
            </label>
            <label class="radio">
              <input type="radio" name="langOption"
                id="csharp" value="C#" />C#
            </label>
            <label class="radio">
              <input type="radio" name="langOption"
                id="erlang" value="Erlang" />Erlang
            </label>
            <label class="radio">
              <input type="radio" name="langOption"
                id="javascript" value="JavaScript" />JavaScript
            </label>
            <label class="radio">
              <input type="radio" name="langOption"
                id="other" value="Other" />Other
            </label>
          </div>
        </fieldset>
        <div class="buttonContainer"><button id="vote" class="btn" />
          Cast Vote</div>
      </form>
    </div>
</div>

<div id="results" class="page" data-role="page" >
    <div id="barChart" class="barChart" data-role="content" >
    </div>
</div>
```

Примечание. Вместо использования общего файла Index.cshtml в обоих представлениях, можно было бы создать новое представление ASP.NET MVC с именем Index.mobile.cshtml и поместить в него разметку для отображения на мобильных устройствах. Этот подход можно использовать, когда необходимо иметь совершенно разное оформление на разных устройствах. Однако, как я полагаю, лучше стремиться иметь общую разметку для обоих представлений и использовать теги, определяющие тип носителя, и другие приемы, позволяющие адаптировать разметку под конкретный тип устройства.

Убедиться, что все работает, можно, переключив параметр `user-agent` браузере. Кроме того, для проверки можно воспользоваться эмулятором и/или симулятором, таким как TestiPhone, чтобы уви-

деть, как будут выглядеть страницы при изменении размеров экрана. Вид страницы в симуляторе iPhone 5 показан на рис. 4.3.

Добавляем поддержку Windows Phone

Способ на основе применения библиотеки jQuery Mobile пригоден для большинства мобильных устройств, однако у нас может появиться желание создать приложения «родные» для мобильной платформы, и воспользоваться в них всеми достоинствами библиотек SignalR. Например, представьте, что у нас появилось желание создать версию нашего приложения опроса для платформы Windows Phone 7. Внешний вид такого приложения на экране устройства показан на рис. 4.4.

Глубокому изучению особенностей создания приложений на языке F# для Windows Phone 7 можно было бы посвятить целую книгу. Однако, чтобы написать простое приложение на F# для Windows Phone 7 совсем не требуется отвлекаться на чтение другой книги.

Рис. 4.3. Внешний вид формы опроса в приложении на основе jQuery Mobile

Рис. 4.4. Внешний вид формы опроса в приложении на основе Windows Phone 7

Просто установите пакет инструментов Windows Phone SDK[1], затем установите один из шаблонов проектов на F# и C# для Windows Phone из галереи Visual Studio Gallery, например, такой как на странице http://bit.ly/TM5gTd.

Я не буду тратить время на обсуждение аспектов данного примера, не связанных с применением библиотек SignalR – полные исходные тексты приложения можно найти на веб-сайте GitHub[2]. Кроме всего прочего отмечу, что этот пример использует автономный сервер SignalR. Реализацию этого сервера можно найти по адресу: http://bit.ly/RWNUob.

> **Примечание.** На момент написания этих строк, инструменты разработки для Windows Phone 7 еще не поддерживались в Visual Studio 2012. Поэтому представленный здесь пример разрабатывался в Visual Studio 2010.

После создания нового проекта F# Windows Phone 7, можно добавить файл XAML[3] (eXtensible Application Markup Language – расширяемый язык разметки для приложений) чтобы оформить приложение, как показано на рис. 4.4. Затем необходимо установить NuGet-пакет `SignalR.Client` в проекты App и AppHost. После этого можно приступать к изменению содержимого файла *AppLogic.fs*, чтобы включить в него поддержку взаимодействий с сервером SignalR.

Сначала определите запись, которая будет играть роль модели данных (`Model`), а также класс, который будет выступать в качестве модели представления (`ViewModel`). Для простоты примера я жестко определил варианты выбора в форме опроса и цвета отображения. Оба определения показаны в следующем примере:

```
type LanguageChoice = { Language : string; BoxColor : string }

type LanguageChoiceViewModel() =
     member x.LanguageChoices =
          let result = List<LanguageChoice>()
          result.Add { Language = "F#"; BoxColor = "#F29925" }
          result.Add { Language = "C#"; BoxColor = "#5492CD" }
          result.Add { Language = "Erlang"; BoxColor = "#E41F26" }
```

[1] http://www.microsoft.com/ru-ru/download/details.aspx?id=27570.
[2] http://bit.ly/XdMPgr.
[3] http://bit.ly/fs-web-cloud.

```
        result.Add { Language = "JavaScript"; BoxColor = "#70BE46" }
        result.Add { Language = "Other"; BoxColor = "#535353" }
        result
```

Примечание. Модель-Представление-МодельПредставления (Model-View-ViewModel, MVVM) – широко известный шаблон проектирования, часто используемый при создании решений на основе XAML. Более подробную информацию о шаблоне проектирования MVVM можно найти в блоге Джона Госсмана (John Gossman), по адресу: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx[1]. Следует отметить, что примеры в этом разделе не точно следуют шаблону MVVM из-за необходимости обеспечить простоту, а также потому, что их цель заключается в демонстрации других аспектов.

Теперь, когда модель данных и модель представления определены, можно изменить класс `MainPage` и включить в него логику, реализующую отправку выбранного варианта на сервер SignalR. Соответствующий код представлен в примере ниже, а дополнительные пояснения к нему приводятся в нескольких следующих абзацах:

```
type MainPage() as this =
    inherit PhoneApplicationPage()
    do this.DataContext <- LanguageChoiceViewModel()
    do Application.LoadComponent(this,
        new System.Uri("/WindowsPhoneApp;component/MainPage.xaml",
            System.UriKind.Relative))

    let confirmationLabel : TextBlock = this?Confirmation

    member this.AnswerButton_Click(sender:obj, e:RoutedEventArgs) =
        let answerButton = sender :?> Button
        SignalR.connection.Send(answerButton.Tag)
            .Start(TaskScheduler.FromCurrentSynchronizationContext())
        confirmationLabel.Text <- "Thanks for Voting!"
```

Первые несколько строк в определении класса связывают его с файлом XAML. Класс наследует `PhoneApplicationPage`, устанавливает нашу модель представления как значение свойства `DataContext`, и, наконец, связывает *MainPage.xaml* с классом.

Следующая строка пытается найти в файле XAML элемент `TextBlock` с именем `Confirmation` и связывает его со значением `confirmationLabel`.

[1] Похожая статья на русском языке: http://msdn.microsoft.com/ru-ru/magazine/dd419663.aspx. – *Прим. перев.*

Эта строка демонстрирует возможности динамического оператора поиска, о котором говорилось выше в этой главе. Шаблон проекта Windows Phone 7 предоставляет реализацию динамического оператора поиска по умолчанию, основной целью которой является поиск ресурсов в коллекции `ResourceDictionary` или элементов управления в файле XAML.

Метод `AnswerButton_Click` демонстрирует традиционную реализацию обработчика события щелчка мыши. Именно здесь мы впервые наблюдаем код, имеющий отношений к библиотеке SignalR. Внутри обработчика сначала определяется, какая кнопка мыши была нажата, затем эта информация отправляется серверу SignalR, за счет запуска задания из текущего контекста синхронизации и, наконец, в метку выводится желаемый текст сообщения подтверждения.

Единственное, что осталось обсудить, – как определяется информация, необходимая для подключения к серверу SignalR. В данном примере это достигается за счет определения модуля с функцией `startConnection`. Эта функция вызывается в момент запуска приложения. И наоборот, когда приложение закрывается, соединение разрывается. Определение модуля с функцией `startConnection` приводится ниже:

```
module SignalR =
    let connection =
        Connection "http://localhost:8081/chartserver"
    let startConnection() =
         if connection.State = ConnectionState.Disconnected then
              connection.Start().Start(
                   TaskScheduler.FromCurrentSynchronizationContext())
```

Объединение F# и NoSQL

Хранилища данных NoSQL в последние годы приобрели особенную популярность. Этот класс хранилищ потеснил традиционные реляционные базы данных, господствовавшие долгое время. В общем случае базы данных NoSQL хранят информацию в неструктурированных или слабоструктурированных форматах, в противоположность строгим схемам, используемым в реляционных базах данных. Эта особенность упрощает горизонтальное масштабирование, обеспечивает избыточность и позволяет обрабатывать гигантские объемы данных за счет удивительно высокой скорости добавления

и извлечения данных. Если вы желаете добиться высокой производительности и масштабируемости решений на F#, которые одинаково хорошо подходят для реализации мобильных и веб-приложений, вам определенно следует больше узнать о базах данных NoSQL и об особенностях взаимодействий с ними из F#.

В этом разделе я покажу, как из программного кода на F# взаимодействовать с некоторыми документо-ориентированными хранилищами NoSQL. Для примера я буду использовать простую таблицу с минимальным объемом информации (имя, фамилия и номер телефона) для списка контактов. Кроме того, в рассматриваемом приложении имеется простая форма, позволяющая добавлять новые контакты. Так как рассматриваемый пример в первую очередь предназначен для демонстрации методов добавления и извлечения записей в/из баз данных NoSQL, я не буду тратить время на аспекты, связанные с реализацией пользовательского интерфейса. Полный комплект исходных текстов примера можно найти по адресу: https://github.com/dmohl/fs-web-cloud-mobile/tree/master/Ch%204/MongoExample.

MongoDB

MongoDB – одно из наиболее популярных решений NoSQL в пространстве .NET. Подобно другим базам данных NoSQL, обсуждаемым в этом разделе, MongoDB является документо-ориентированным хранилищем. То есть, информация в базе данных хранится в виде ключа и связанного с ним документа. Когда позднее выполняется обращение к документу по ключу, извлечение происходит исключительно быстро. Инструкции по установке и настройке MongoDB в Windows доступны по адресу: http://bit.ly/mongodb-windows.

Когда все этапы установки, настройки и запуска MongoDB будут пройдены, следующим шагом становится создание клиента, взаимодействующего с этой базой данных. Проще всего организовать работу с MongoDB из F# – установить NuGet-пакет `MongoFs`. В результате в проект будет добавлен официальный клиент MongoDB на C#, а также простая обертка на F#, которая немного упростит взаимодействие с клиентским API MongoDB на C# из F#.

Как упоминалось выше, я представлю простой пример для каждого из трех документо-ориентированных хранилищ, обсуждаемых в этом разделе. Каждый пример – это проект ASP.NET MVC 4, созданный из шаблона C#/F# ASP.NET MVC 4, рассматривавшегося

в главе 1. В примере используется запись `Contact`, объявление которой приводится ниже:

```
namespace FsWeb.Models

open MongoDB.Bson
open System.ComponentModel.DataAnnotations

[<CLIMutable>]
type Contact = {
    _id : ObjectId
    [<Required>] FirstName : string
    [<Required>] LastName : string
    [<Required>] Phone : string
}
```

В этом коде есть два аспекта, требующих дополнительных пояснений. Первый аспект – тип `ObjectId`. `ObjectId` – это 12-байтный двоичный тип, поддерживаемый хранилищем MongoDB. Значения этого типа обычно используются для представления уникальных идентификаторов документов.

Второй аспект – атрибут `CLIMutable`, который периодически встречался нам во всех предыдущих главах. Этот атрибут является малоизвестной особенностью F# 3.0, имеющей большое значение при работе с записями, которые должны преобразовываться в последовательную форму и обратно, что обычно происходит при передаче документа в хранилище или извлечении его оттуда. Когда компилятор F# встречает этот атрибут, он автоматически добавляет конструктор без параметров и методы чтения/записи для свойств. Этот малозаметный атрибут открывает возможность использовать записи там, где раньше это было весьма затруднительно, например, в контроллерах приложений на основе ASP.NET MVC.

В следующем примере приводится код, извлекающий все контакты, хранящиеся в базе данных MongoDB, а также код, создающий новые контакты:

```
namespace FsWeb.Controllers

open System.Linq
open System.Web
open System.Web.Mvc
```

```
open FsWeb.Models

[<HandleError>]
type HomeController() =
    inherit Controller()

    let contacts =
        createLocalMongoServer()
        |> getMongoDatabase "contactDb"
        |> getMongoCollection "contacts"

    member this.Index () =
        contacts.FindAll().ToList() |> this.View

    [<HttpGet>]
    member this.Create () =
        this.View()

    [<HttpPost>]
    member this.Create (contact:Contact) =
        if base.ModelState.IsValid then
            contact |> contacts.Insert |> ignore
            this.RedirectToAction("Index") :> ActionResult
        else
            this.View() :> ActionResult
```

Выделенный код показывает, как мало нужно для взаимодействия с MongoDB из F# при использовании `MongoFs`. Первые четыре выделенные строки идентифицируют экземпляр MongoDB, базу данных и коллекцию, с которой будут выполняться операции. Кроме того, если база данных или коллекция еще не существует, она будет создана. Используемый здесь прием с использованием конвейерного оператора обеспечивается библиотекой `MongoFs`. Всего одна строка потребовалась, чтобы извлечь все записи, и так же всего одна строка потребовалась, чтобы добавить новую запись – это возможно исключительно благодаря синтаксису языка F#, в комбинации с API официального клиента MongoDB на C#.

Возможно, кто-то заметил, что в этом примере я не открываю модуль `MongoFs`. Это возможно благодаря еще одной особенности языка F# в виде атрибута `AutoOpen`. Когда в модуль добавляется атрибут `AutoOpen`, этот модуль не требуется открывать или ссылаться на него явно. Ниже приводится пример использования атрибута `AutoOpen`:

```
[<AutoOpen>]
Module MongoFs

// Код опущен ради экономии места
```

RavenDB

RavenDB – еще одна документо-ориентированная база данных, вихрем ворвавшаяся в этот мир. В число больших преимуществ RavenDB входят: поддержка транзакций, полнотекстовый поиск (full-text search) с помощью библиотеки Lucene, основной принцип «безопасность по умолчанию» и простота создания встроенной базы данных.

Чтобы опробовать пример, следующий ниже, загрузите и распакуйте последнюю версию RavenDB. Запустите сервер RavenDB. После этого вы сможете выполнить пример. Данный пример создавался и опробовался с версией 1.0.960 NuGet-пакета `RavenDB.Client.FSharp`, установленной в решение F#/C# ASP.NET MVC. Код, запускающий пример и извлекающий список контактов, показан в следующем примере:

```
namespace FsWeb.Controllers

open System.Linq
open System.Web
open System.Web.Mvc
open FsWeb.Models
open Raven.Client.Document

[<HandleError>]
type HomeController() =
    inherit Controller()

    let ravenUrl = "http://localhost:8080/"

    let executeRavenAction action =
        use store = new DocumentStore()
        store.Url <- ravenUrl
        store.Initialize() |> ignore
        use session = store.OpenSession()
        action session

    member this.Index () =
```

```
        executeRavenAction
        <| fun session -> session.Query<Contact>().ToList()
        |> this.View

    [<HttpGet>]
    member this.Create () =
        this.View()

    [<HttpPost>]
    member this.Create (contact:Contact) =
        if base.ModelState.IsValid then
            executeRavenAction
            <| fun session ->
                session.Store contact
                session.SaveChanges()
            this.RedirectToAction("Index") :> ActionResult
        else
            this.View() :> ActionResult
```

Хотя код получился более объемным, чем в примере MongoDB, он дает несколько ключевых преимуществ, таких как врожденная поддержка шаблона проектирования «Единица работы» (Unit of Work). Помимо клиентского API RavenDB на F#, предыдущий фрагмент демонстрирует также функцию высшего порядка `executeRavenAction`, которая абстрагирует общий код, используемый для выполнения большинства операций с RavenDB.

CouchDB

Последнее решение NoSQL, которое я продемонстрирую, – база данных CouchDB. Написанная на языке Erlang, база данных CouchDB является одним из самых надежных решений NoSQL. Это очень производительная база данных, которая к тому же проста в распространении. Найти более подробную информацию о CouchDB можно по адресу: http://couchdb.apache.org/.

Примечание. Создатель CouchDB в настоящее время основное внимание уделяет разработке проекта базы данных с именем Couchbase, которая имеет иной API и множество дополнительных особенностей. В этой книге я представлю только базу данных CouchDB. Однако вам стоит взглянуть на Couchbase, чтобы выяснить, подходит ли она для вас. Сравнение CouchDB b Couchbase можно найти по адресу: http://vschart.com/compare/couchdb/vs/couchbase.

Существует множество клиентских пакетов, обеспечивающих возможность взаимодействий с базой данных CouchDB. Один из таких пакетов, созданный мной, называется `FSharpCouch`. Чтобы задействовать его, установите NuGet-пакет `FSharpCouch` в требуемый проект и используйте функции, описанные по адресу: https://github.com/dmohl/FSharpCouch. Ниже приводится короткий пример операций с тем же списком контактов, что и в примерах выше, где применялись другие решения NoSQL:

```
namespace FsWeb.Controllers

open System.Linq
open System.Web
open System.Web.Mvc
open FsWeb.Models
open FSharpCouch

[<HandleError>]
type HomeController() =
     inherit Controller()

     let couchUrl = "http://localhost:5984"
     let dbName = "people"

     member this.Index () =
          getAllDocuments<Contact> couchUrl dbName
          |> Seq.map(fun c -> c.body)
          |> this.View

     [<HttpGet>]
     member this.Create () =
          this.View()

     [<HttpPost>]
     member this.Create (contact:Contact) =
          if base.ModelState.IsValid then
              contact |> createDocument couchUrl dbName |> ignore
              this.RedirectToAction("Index") :> ActionResult
          else
              this.View() :> ActionResult
```

Один интересный аспект этой реализации состоит в том, что в процессе работы она не изменяет записи. Запись `Contact`, исполь-

зуемая для создания запроса, может включать только самое необходимое – имя, фамилию и номер телефона. Операции создания и извлечения документов возвращают другую запись, содержащую ключ (ID) документа и номер его версии, как показано ниже, с полем `body`, содержащим заполненный экземпляр, соответствующий первоначально добавленной записи (это основная причина вызова `Seq.map` в методе `Index`, в примере выше):

```
type CouchDocument<'a> = {
     id : string
     rev : string
     body : 'a
}
```

В заключение

В этой главе были рассмотрены некоторые дополнительные приемы создания более производительных и масштабируемых мобильных и веб-приложений на языке F# и другие технологии. Вы познакомились с несколькими различными способами создания и использования веб-сокетов (Web Sockets), а также с несколькими представителями документо-ориентированных хранилищ данных. На протяжении всей главы я описывал некоторые особенности F#, с которыми вы, возможно, прежде не были знакомы.

В следующей главе я познакомлю вас с несколькими возможностями создания пользовательских интерфейсов в элегантном функциональном стиле. Они позволят вам создавать полноценные стеки с применением функциональных концепций дадут вам пищу для новых мыслей и идей. Вы не должны упустить такой шанс!

Глава 5. Разработка интерфейсов в функциональном стиле

Искусство бросает вызов технологиям,
а технологии вдохновляют искусство.

– Джон Лассетер (John Lasseter)

Сколько я себя помню, я играл на гитаре. Я рос, путешествуя по выходным с моими родителями, и играл в их музыкальной группе. Это привило мне любовь к сочетанию «науки» (то есть, теории музыки и так далее) и искусства.

> **Примечание.** Найти мои музыкальные сочинения можно по адресу: http://www.danielmohl.com/.

Когда я начинал работать с языками программирования, у меня возникали похожие ощущения. Программирование имеет множество аспектов, укладывающихся в определение «наука», однако процесс создания простого и красивого кода – это определенно искусство.

На протяжении всей книги я показывал вам различные способы создания облачных, мобильных и веб-приложений на языке F#. В большинстве случаев эти способы относились к созданию серверной части приложений. С каждым примером я все больше и больше увеличивал использование функциональных концепций, и реализация этих концепций делала приложения более элегантными и красивыми. Было бы здорово иметь возможность использовать эти же концепции для разработки пользовательских интерфейсов в функциональном стиле.

Как все мы знаем, JavaScript является основным языком программирования, применяемым для создания клиентской части веб-приложений. Мне действительно очень нравится JavaScript, особенно его функциональные стороны. Однако было бы еще лучше, если бы он имел еще больше функциональных особенностей. Существует

ли какой-нибудь способ сделать JavaScript более функциональным? Да, конечно! Существует множество библиотек[1], статей[2] и книг[3], рассказывающих о реализации типичных функциональных концепций на JavaScript.

Одним из недавно появившихся направлений в веб-разработке стало использование различных инструментов, осуществляющих компиляцию программного кода на других языках в программный код на JavaScript. Эти инструменты дают возможность использовать функциональные языки для создания кода на JavaScript в ваших приложениях.

В этой главе я немного отклонюсь от основного направления предыдущих глав и познакомлю вас с некоторыми средствами, позволяющими писать клиентский код для веб-приложений в функциональном стиле, не все из которых имеют отношение к языку F#. Поскольку приемы использования языка JavaScript можно найти в других книгах, я сосредоточусь исключительно на компиляторах.

Подготовка почвы

На протяжении этой главы мы будем следовать общему шаблону. Сначала я представлю язык программирования и/или инструмент. Затем расскажу о некоторых преимуществах данного инструмента, дам информацию о том, как начать пользоваться им, и в заключение представлю пример создания приложения.

Каждый из трех языков/инструментов, о которых пойдет речь в этой главе, имеет множество достоинств. Определение «лучшего» во многом зависит от решаемых вами задач, от команды разработчиков, от архитектурных целей разработки и от личных предпочтений.

Роль примера, на котором будут демонстрироваться различия между разными подходами, сыграет простое приложение «To Do», позволяющее создавать списки дел и отмечать ход их выполнения. Это далеко не полная реализация, но ее вполне достаточно, чтобы показать различные аспекты рассматриваемых инструментов. Главная страница приложения отображает две группы заданий (дел) – группа заданий, которые требуется выполнить, и группа выполненных заданий. Выполнив задание, его можно отбуксировать мышью

---

[1] http://ibm.co/U2iuO3.
[2] http://bit.ly/ben-alman.
[3] http://bit.ly/udon-javascript.

из колонки **To Do** (Что сделать) в колонку **Done** (Выполнено). Внешний вид приложения показан рис. 5.1.

Рис. 5.1. Внешний вид приложения To Do

В дополнение к коду на JavaScript, сгенерированному соответствующим компилятором, во всех примерах будут использоваться библиотеки jQuery и jQuery UI. В примерах также используются определения стилей CSS из jQuery UI и Twitter Bootstrap. Итак, продолжим!

Знакомство с LiveScript

Если вы в веб-разработке не первый год, первая мысль, которая может посетить вас, когда вы услышите слово *LiveScript*, что это прежнее название JavaScript. Хотя раньше хорошо известный язык клиентских сценариев действительно назывался LiveScript, в настоящее время это имя получило вторую жизнь и обозначает язык, программный код на котором компилируется в код на JavaScript.

На главном веб-сайте LiveScript[1] говорится, что этот язык «является производным от языка Coco, который в свою очередь является производным от CoffeeScript». Он унаследовал все самое лучшее из этих двух языков и привнес новые особенности, лучше подходящие для разработки в функциональном стиле.

Преимущества

Язык LiveScript обладает множеством преимуществ, но я остановлюсь лишь на тех, что имеют прямое отношение к языку F#. Хотя синтаксис LiveScript отличается от синтаксиса F#, как и другие

[1] http://gkz.github.io/LiveScript/.

варианты, рассматриваемые в этой главе, он поддерживает многие концепции и обладает многими свойствами, по своему духу близкими к концепциям и свойствам F#.

> **Примечание.** Следующий перечень особенностей LiveScript не претендует на полноту. Полный их список, а также массу примеров можно найти на веб-сайте LiveScript[1], а также в статьях в Интернете[2].

Первая особенность, имеющая отношение к языку F# – значимые пробелы. В языке LiveScript, как и в F#, пробелы используются для выделения блоков кода. Кроме того, запятые, круглые, фигурные и квадратные скобки часто являются необязательными. Это делает код не только более компактным и выразительным, но и читаемым.

Как упоминалось выше, LiveScript обладает разными особенностями, присущими функциональным языкам и сходными с аналогичными особенностями F#. В том числе: каррированные функции, частичное применение функций, простой механизм сопоставления с образцом, прямой и обратный конвейерные операторы (`|>` и `<|`), и операторы композиции функций (`>>` и `<<`). Библиотека `prelude.ls` – написанная автором языка LiveScript – добавляет различные функции высшего порядка, такие как `map`, `filter`, `head`, `tail`, `fold`, `cons`, `append`, `zip` и многие другие.

Так как LiveScript ведет свою родословную от CoffeeScript, девизом которого является фраза: «Это всего лишь JavaScript», – навыки программирования на JavaScript могут оказаться отличным фундаментом для начинающих осваивать LiveScript. Кроме того, код на JavaScript, который производит компилятор, получается весьма удобочитаемым. Эти два аспекта языка существенно упрощают отладку сценариев и обеспечивают бесшовную интеграцию с с другими библиотеками на JavaScript.

Применение

Самый простой способ приступить к использованию языка LiveScript – установить его с помощью диспетчера пакетов Node Package Manager (например, `npm install -g LiveScript`). Можно также создать копию репозитория на GitHub и установить LiveScript из копии. Разумеется, при этом подразумевается, что на компьютере должна

---

[1] http://gkz.github.io/LiveScript/.

[2] http://bit.ly/programming-javascript и http://bit.ly/javascript-part2.

быть установлена виртуальная машина Node.js. Если прежде вам не приходилось использовать Node.js, не пугайтесь. Установить ее проще простого. Посетите веб-сайт Node.js, где можно найти дополнительную информацию[1].

> **Примечание.** Если вам интересно поэкспериментировать с LiveScript, но вы не желаете связываться с установкой и настройкой, посетите веб-страницу LiveScript на сайте GitHub[2], где вы найдете веб-версию интерактивной оболочки REPL.

Установив LiveScript, можно создать первый файл для последующей компиляции. Файлы с кодом на LiveScript имеют расширение *.ls* и а их имена будут использоваться для создания файлов с кодом на JavaScript. В данном примере мы дадим файлу имя *test.js.ls*. Создайте этот файл и добавьте в него следующий код:

```
weekend = <[ sat sun ]>

isWeekend = (dayOfWeek) ->
    | dayOfWeek in weekend => true
    | _ => false

'sun' |> isWeekend |> console.log
```

Большая часть кода должна показаться вам очень знакомой, так как синтаксис LiveScript достаточно близок к синтаксису F#. Первая строка в этом сценарии создает массив строк. Следующие три строки определяют функцию с именем `isWeekend`, принимающую единственный аргумент. Оператор сопоставления с образцом в этой функции определяет, содержится ли указанная строка в массиве `weekend`. В конце используется прямой конвейерный оператор, чтобы выяснить, является ли строка `"sun"` частью массива `weekend`, и передать результат функции `console.log`.

Чтобы скомпилировать этот код в код на JavaScript, выполните следующую команду в командной оболочке (где *src/* – имя каталога, содержащего файлы *.ls*, а *lib/* – каталог, куда должны сохраняться скомпилированные файлы):

```
livescript --compile --output lib/ src/
```

[1] http://nodejs.org/.
[2] http://gkz.github.io/LiveScript/.

Получившиеся в результате файлы JavaScript можно использовать точно так же, как любые другие файлы JavaScript.

Пример

Теперь, когда я показал вам очень простой пример, можно попробовать воспользоваться всеми нашими знаниями и создать чуть более сложный и чуть более практичный пример. Это будет первый действующий пример приложения To Do с поддержкой буксировки элементов мышью, о котором я рассказывал выше. Я уже показывал, как будет выглядеть пользовательский интерфейс приложения, поэтому окунемся сразу в код.

Чтобы дать некоторое представление, ниже приводится код разметки, с которой мы будем работать:

```
<div role="main">
     <div class="container todoList">
          <h1>My To Do List</h1>
          <div class="detail row-fluid">
               <div class="span6 taskContainer well">
                    <div class="column-header">
                         <h3>Not Started</h3>
                    </div>
                    <div class="tasks tasksNotStarted droppable">
                    </div>
               </div>
               <div class="span6 taskContainer well">
                    <div class="column-header">
                         <h3>Done</h3>
                    </div>
                    <div class="tasks tasksDone droppable"/>
               </div>
          </div>
     </div>
</div>
```

Ниже приводится код на LiveScript, который приводит интерфейс в движение:

```
let $ = jQuery
     # Добавить каждое задание в соответствующий элемент DIV
     populateTaskList = ( $el, taskList ) ->
          taskList
```

```
        |> prelude.each ( (t) ->
             $ "<div class='ui-widget-content draggable'>#{t}</div>"
             .appendTo $el )

    # Инициализация
    do ->
         tasksToDo = [ "Persist the tasks to a data store."
              "Add new tasks."
              "Remove a task." ]
         tasksDone = [ "Allow tasks to be moved to done."
              "Add dynamic population of tasks." ]

         populateTaskList $( ".tasksNotStarted" ), tasksToDo
         populateTaskList $( ".tasksDone" ), tasksDone

    # Определить буксируемые элементы.
    $ ".draggable" .draggable (
         revert: "invalid"
         cursor: "move"
         helper: "clone"
    )

    # Настроить области "сброса"
    $ ".droppable" .droppable (
        hoverClass: "ui-state-active"
        accept: ".draggable"
        drop: ( event, ui ) -> ui.draggable.appendTo $ @
    )
```

Примечание. Символ # в LiveScript отмечает начало комментариев.

Давайте пройдемся через этот код. Первая строка обертывает весь оставшийся код в самовызывающуюся анонимную функцию. Кроме того, она гарантирует сохранение в переменной `$` ссылки на объект `jQuery`. Это делается на тот случай, если какой-то другой сценарий изменит значение `$`. Оба эти приема относятся к рекомендуемой практике в мире JavaScript.

Далее определяется функция с именем `populateTaskList`. Эта функция принимает два параметра. Первый представляет элемент, куда будут добавляться задания, а второй – массив строк, представляющих задания. Затем массив заданий передается функции `each` из библиотеки `prelude.ls`, чтобы для каждого задания создать элемент `div` в указанном контейнере.

Примечание. Вместо `prelude.each` в этом примере можно было бы использовать функцию `jQuery.each`, устранив тем самым одну зависимость. Однако, библиотека `prelude.ls` включает множество очень полезных функций, которые отсутствуют в библиотеке jQuery, поэтому я предпочел оставить эту зависимость и показать, как можно пользоваться библиотекой `prelude.ls`.

Далее определяется самовызывающаяся анонимная функция, которая просто создает два массива и передает их другой функции для обработки. Первый массив содержит строки, определяющие еще не выполненные задания, а второй – строки с выполненными заданиями. Каждый массив затем передается функции `populateTaskList`, вместе со ссылкой на элемент, куда должны быть помещены списки заданий.

Последние два важных аспекта в этом сценарии связаны с определением буксируемых элементов и областей «сброса». Для этой цели используется стандартный API библиотеки jQuery UI. Здесь следует остановиться на двух моментах. Во-первых, возможно вы отметили отсутствие в коде некоторых символов, таких как скобки, запятые и точки с запятой. За возможность опустить скобки (как, например, в выражении `$ ".draggable"`) можно поблагодарить язык Coco, который, как уже говорилось, является прародителем LiveScript. Возможность опускать запятые и точки с запятой унаследована от CoffeeScript. Это – наглядные примеры возможности опускать некоторые необязательные символы, упоминавшейся выше. Второй момент, который следует отметить, – использование символа `@` в определении функции, вызываемой по событию сброса. Символ `@` – это всего лишь более короткий псевдоним ключевого слова `this` в JavaScript.

Язык LiveScript дает отличную возможность оставаться достаточно близко к JavaScript, и при этом писать код в более функциональном стиле. Он прозрачно интегрируется с любыми другими библиотеками на JavaScript и, несмотря на свою молодость, является весьма стабильным, благодаря хорошей наследственности. Полные исходные тексты этого примера можно найти по адресу: http://bit.ly/livescript.

Исследуем Pit

Несмотря на то, что язык LiveScript дает отличную возможность писать клиентский код для веб-приложений в функциональном стиле, он все-таки не является языком F#. Было бы здорово, если

бы имелась возможность использовать настоящий язык F# для создания клиентских сценариев. Такую возможность может дать нам компилятор Pit.

Компилятор Pit был создан Фахадом Сухаибом (Fahad Suhaib) и Картиком Вишну (Karthik Vishnu), на основе языка F#. Впервые он был размещен на сайте GitHub в декабре 2011, и, поскольку этот проект не имеет предшественников, он является самым молодым, из рассматриваемых в этой главе.

Давайте посмотрим, какие выгоды может предложить компилятор Pit.

Преимущества

Pit поддерживает два разных способа запуска компиляции. Первый – вызов компилятора Pit с параметрами из командной строки (подобно тому, как мы компилировали код на LiveScript). Второй – использование интеграции с Visual Studio или MonoDevelop для создания проекта, которая сама позаботится о вызове компилятора с нужными командами. Если прежде вам приходилось пользоваться Visual Studio и/или MonoDevelop, второй способ будет для вас намного проще. Это также означает отсутствие каких-либо внешних зависимостей, например, от Node.js.

Самое большое преимущество компилятора Pit перед другими альтернативами, такими как LiveScript, состоит в том, что он компилирует код на F#. Это здорово, потому что позволяет использовать все знания языка F# и применять их при создании клиентских сценариев для веб-приложений. Полный перечень особенностей языка F#, поддерживаемых компилятором Pit, можно найти по адресу: http://bit.ly/pitfw-docs.

Последнее преимущество компилятора Pit, о котором мне хотелось бы упомянуть, касается возможности отладки. Компилятор Pit поддерживает два режима компиляции: с отладочной информацией (debug) и без нее (release). В режиме компиляции с отладочной информацией, Pit использует библиотеку времени выполнения Silverlight наряду с мостом Silverlight-to-JavaScript, что обеспечивает удобную возможность отладки. Так как большинство альтернативных инструментов, осуществляющих компиляцию кода на высокоуровневых языках в код на JavaScript, не поддерживают возможность отладки, это обстоятельство может превратиться в большое преимущество. Более подробную информацию об этой особенности можно найти в документации, упомянутой в предыдущем абзаце.

Примечание. Решить проблему отладки в различных инструментах, компилирующих программный код на других языках в JavaScript, призвана новая особенность, называющаяся «отображение исходных текстов». Увидеть пример ее использования можно по адресу: http://www.youtube.com/watch?v=-xJl22Kvgjg.

Применение

Самый простой способ приступить к использованию компилятора Pit в Windows – загрузить дистрибутив[1], извлечь файл *Pit.Setup.msi* запустить установку. Мастер установки сохранит выполняемый файл компилятора Pit (*pfc.exe*) и библиотеки поддержки в каталог *Pit*, 32-битной версии каталога *program files*. Кроме того, в процессе установки будет запущено расширение для Visual Studio, которое добавит несколько новых шаблонов проектов, чтобы упростить использование компилятора в Visual Studio 2010.

Обратите внимание: в предыдущем абзаце говорится об установке расширения Pit в Visual Studio 2010. На момент написания этих слов, компилятор Pit поддерживал только Visual Studio 2010, однако реализация поддержки Visual Studio 2012, как множества других интересных особенностей, уже включена в планы. В настоящее время есть два способа использовать Pit в Visual Studio 2012. Первый – просто вызывать его из командной строки. Второй – вручную отредактировать пакет Visual Studio eXtension (VSIX), добавляющий шаблоны проектов, чтобы он распознавал Visual Studio 2012. Для этого нужно переименовать файл пакета VSIX, изменив расширение на *.zip*, распаковать архив, изменить элемент `SupportedProducts` в файле *extension.vsixmanifest*, чтобы он выглядел, как показано ниже, снова сжать все файлы в архив и дать ему расширение *.vsix*. После этого можно выполнить расширение VSIX, чтобы установить шаблоны проектов вVisual Studio 2012:

```
<SupportedProducts>
  <VisualStudio Version="11.0">
    <Edition>Ultimate</Edition>
    <Edition>Premium</Edition>
    <Edition>Pro</Edition>
  </VisualStudio>
  <VisualStudio Version="10.0">
```

[1] http://pitfw.org/.

```
    <Edition>Ultimate</Edition>
    <Edition>Premium</Edition>
    <Edition>Pro</Edition>
  </VisualStudio>
</SupportedProducts>
```

Примечание. Дополнительную информацию о создании расширений для Visual Studio на языке F# можно найти в статье на сайте MSDN[1].

Необходимо сделать еще кое-что, чтобы получить возможность собрать пример в следующем разделе. Как уже говорилось, компилятор Pit продолжает активно развиваться. В числе последних улучшений – поддержка библиотек jQuery и jQuery UI. Чтобы воспользоваться преимуществами этих улучшений, следует загрузить последние стабильные сборки (версия 0.3.3, на момент написания этих слов) на странице загрузки Pit[2].

Загрузив архив, извлеките из него файлы и перепишите их поверх существующих библиотек, выполняемого файла и файлов JavaScript (которые находятся в каталоге установки, например: *C:\Program Files (x86)\Pit\0.2\bin*). Теперь, когда все подготовительные операции по установки выполнены, можно приступить к сборке нашего приложения To Do.

Внимание. Не удаляйте файлы из каталога C:\Program Files (x86)\Pit\0.2\bin перед копированием новых файлов.

Пример

Прежде чем запустить сборку приложения To Do, создайте проект из шаблона ASP.NET MVC 4. Затем добавьте в проект ссылки на файлы *Pit.Core.js* и *Pit.js* из каталога *bin* в пути установки Pit, как на любые другие файлы JavaScript. И, наконец, добавьте новый проект, создав его из шаблона Pit Library, который можно найти в разделе **Visual F# → Pit**, в диалоге мастера создания проекта.

Примечание. Несмотря на то, что пример использует фреймворк ASP.NET MVC, в нем отсутствует код, обращающийся к этому фреймворку. С таким же успехом можно было бы создать только проект Pit Library и

[1] http://msdn.microsoft.com/ru-ru/magazine/hh456399.aspx.
[2] http://bit.ly/pitfw-download.

использовать старую добрую разметку HTML, таблицы стилей CSS и так далее, для всего остального.

Теперь осталось лишь настроить автоматическое копирование сгенерированных файлов JavaScript из выходного каталога проекта Pit Library в каталог *Scripts* веб-проекта. Я реализовал это так:

```
copy /Y "$(TargetDir)$(ProjectName).js"
  "$(SolutionDir)PitTodoExample\PitTodoExampleWeb\Scripts\$(ProjectName).js"
```

Компилятор Pit создаст файл JavaScript, необходимый для работы приложения, а мы должны добавить файл *.fs* в проект Pit Library. Код, который следует сохранить в этом файле, представлен ниже:

```
namespace PitTodo

open Pit
open Pit.Dom
open Pit.JavaScript
open Pit.JavaScript.JQuery
open Pit.JavaScript.JQuery.UI

module mainModule =
    [<JsObject>]
    type dragType = { draggable : DomElement }

    [<Js>]
    let populateTaskList (el:DomElement) tasksList =
        tasksList
        |> List.iter( fun (task:string) ->
            tag "div" [|
                "class"@="ui-widget-content draggable"
                "innerHtml"@=task |]
            |> Html.make
            |> el.AppendChild )

    [<Js>]
    let populateTasks () =
        let tasksToDo =
            [ "Persist the tasks to a data store."
              "Add new tasks."
              "Remove a task." ]
        let tasksDone =
```

```
            [ "Allow tasks to be moved to done."
              "Add dynamic population of tasks." ]
        let tasksNotStartedEl = document.QuerySelector ".tasksNotStarted"
        let tasksDoneEl = document.QuerySelector ".tasksDone"
        populateTaskList tasksNotStartedEl tasksToDo
        populateTaskList tasksDoneEl tasksDone

    [<Js>]
    let initDragAndDrop () =
        jQueryUI(".draggable")
            .draggable(
                [ "revert" => "invalid"
                  "cursor" => "move"
                  "helper" => "clone" ] )
            .ignore()

        jQueryUI(".droppable")
            .droppable(
                [ "hoverClass" => "ui-state-active"
                  "accept" => ".draggable"
                  "drop" => fun (event, ui:dragType) ->
                      jQuery("this").append(ui.draggable).ignore() ] )
            .ignore()

    [<DomEntryPoint>]
    [<Js>]
    let main() =
        populateTasks()
        initDragAndDrop()
```

Первое, что бросается в глаза, – применение различных атрибутов к функциям и/или типам. В данном приложении используются три наиболее обычных атрибута. Атрибут `JsObject` сообщает компилятору, что данный тип записи должен компилироваться в литерал JavaScript. Атрибут `Js` указывает, что функция, конструктор или метод на языке F# должен компилироваться в код на JavaScript. Атрибут `DomEntryPoint` определяет точку входа в приложение.

> **Примечание.** Атрибуты, распознаваемые компилятором Pit, такие как `Js`, – это лишь частные случаи, или псевдонимы стандартного объекта `ReflectedDefinitionAttribute` в F#, возвращающего код не в виде инструкций на промежуточном языке IL, а в виде абстрактного синтаксического дерева.

Сам код мало отличается от версии на языке LiveScript, разве что вы получаете поддержку строгого контроля типов, свойственного языку F#. В первую очередь в этой реализации определяется запись `dragType` и снабжается атрибутом, обеспечивающим ее компиляцию в литерал объекта JavaScript. Она будет использоваться ниже, когда требуемые элементы будут определяться как доступные для сброса при буксировке их мышью. Далее определяется функция `populateTaskList`, создающая элементы `div` для каждой строки в переданном ей списке строк, и добавляющая их в указанный элемент. Функция `populateTasks` решает те же задачи, что и одноименная функция в примере на LiveScript. Следующая функция, `initDragAndDrop`, настраивает указанные элементы, как пригодные для буксировки или сброса. Наконец, функция `main` запускает весь механизм в действие, вызывая `populateTasks` и `initDragAndDrop`.

> **Примечание.** Если взглянуть на код JavaScript, созданный компилятором Pit, можно увидитеть одно большое отличие от версии, полученной в результате компиляции сценария на LiveScript. Если в коде JavaScript, полученном из LiveScript в основном используются самовызывающиеся анонимные функции, чтобы предотвратить загрязнение глобального пространства имен, то Pit использует шаблон пространства имен. Оба приема одинаково пригодны для достижения поставленной цели. Подробнее узнать об этих и других шаблонах программирования можно в книге Стояна Стефанова (Stoyan Stefanov) «JavaScript Patterns» (O'Reilly)[1].

Полные исходные тексты примера приложения Pit To Do можно найти по адресу: http://bit.ly/pittodo.

Погружение в WebSharper

Наиболее известным фреймворком для создания веб-приложений на языке F#, является WebSharper. Из всех вариантов, представленных в этой главе, WebSharper является самым зрелым и проверенным в промышленных условиях. На основе фреймворка WebSharper создан один из известных сайтов – FPish[2], являющийся представительством Интернет-сообщества функциональных программистов. Фреймворк WebSharper также используется многими известными компаниями, включая Microsoft и Ford.

---

[1] Стоян Стефанов, «JavaScript. Шаблоны», ISBN: 978-5-93286-208-7, Символ-Плюс, 2011. – *Прим. перев.*

[2] http://fpish.net/.

Как и Pit, WebSharper компилирует код на F# в код на JavaScript. Однако он не ограничивается компиляцией в файлы JavaScript. Он предоставляет также возможность создавать полноценные веб-стеки на F#. Это означает, что помимо клиентского и серверного кода имеется также возможность писать разметку на F#. Кроме того, фреймворк WebSharper делает реализацию взаимодействий между клиентом и сервером тривиальным делом.

Давайте познакомимся и с другими его преимуществами.

Преимущества

Как уже упоминалось выше, одним из самых больших преимуществ фреймворка WebSharper является возможность создавать весь комплекс программных компонентов на F#. Это устраняет необходимость переключаться с одного языка на другой, использовать разный синтаксис, концепции и идиомы. Так как для создания всех программных компонентов выполняется на языке F#, вы автоматически получаете все его преимущества, такие как безопасность типов, механизм определения типов, стандартные модули, такие как `Seq`, и множество различных инструментов, входящих в состав Visual Studio.

Фреймворк WebSharper продолжает развиваться, оставаясь на острие технологических тенденций. В него постоянно вносятся новые особенности, которые можно использовать, даже не являясь экспертом в том или ином направлении развития технологий. Например, фреймворк WebSharper широко использует новые возможности стандарта HTML5, но вам не нужно быть экспертом в HTML5, чтобы пользоваться ими. Кроме того, WebSharper берет на себя все хлопоты по обработке различий между браузерами, автоматически определяет зависимости, необходимые для страницы и загружает только их, а также предоставляет доступ к различным популярным библиотекам через расширения WebSharper. Мобильные приложения на основе WebSharper могут создаваться с применением расширений для таких библиотек, как jQuery Mobile, SenchaTouch и Kendo UI, а также настроек, позволяющих упаковывать эти приложения в «родной» формат приложений для Android или Windows Phone 7. Другой пример, подтверждающий сказанное выше, – поддержка Visual Studio 2012 была включена в WebSharper еще до выхода коммерческой версии этой среды разработки.

Примечание. Директор компании IntelliFactory, Адам Гранич (Adam Granicz), обсуждает некоторые абстракции, предлагаемые фреймворком WebSharper, и приводит пример мобильного приложения в статье для веб-сайта InfoQ[1].

Конечная цель фреймворка WebSharper состоит в том, чтобы помочь вам еще быстрее создавать мобильные и веб-приложения. Я коснулся лишь вершины айсберга возможностей фреймворка WebSharper, позволяющих достичь этой цели, и настоятельно советую ознакомиться с примерами использования WebSharper по адресу: http://websharper.com/SamplesPage.

Применение

Чтобы получить возможность задействовать фреймворк WebSharper, достаточно установить один или более компонентов и затем использовать привычный уже механизм создания новых проектов в Visual Studio. На странице загрузки WebSharper[2] можно загрузить сам фреймворк (на момент написания этих строк самой свежей была версия WebSharper 2.4 Q2), а также различные расширения для него. Кроме того, в NuGet можно найти разнообразные пакеты для WebSharper.

Как только все будет установлено, в мастере создания проектов, в категории **Visual F#**, появится новая подкатегория **WebSharper**. Выбрав эту подкатегорию, вы увидите список шаблонов проектов для ASP.NET, ASP.NET MVC, Android и сайтлетов (Sitelets). В дополнение к шаблонам, содержащим только код, необходимый для начала разработки новых решений на основе WebSharper, вы найдете также шаблоны с примерами законченных приложений. Они окажут большую помощь начинающим осваивать WebSharper.

Примечание. Сайтлеты (sitelets) – это совершенно иной подход к созданию приложений на основе шаблона проектирования MVC. Как отмечается на веб-сайте WebSharper[3], они являются «удобной абстракцией компонентов веб-сайта. Сайтлеты – это наиболее правильная реализация MVC с применением действий первого рода (first-class actions) и перекрестных ссылок».

---

[1] http://www.infoq.com/articles/WebSharper.
[2] http://websharper.com/downloads.
[3] http://websharper.com/home.

Например, шаблон проекта Sample Web Application (Sitelets) генерирует приложение, включающее примеры использования формлетов (formlets), механизма шаблонов, маршрутизации, композиции сайтлетов, интеграции со страницей ASPX, не связанной с сайтлетами непосредственно, и многие другие. Более подробную информацию о сайтлетах и формлетах можно найти на странице http://websharper.com/docs/abstractions[1].

В Интернете можно также найти множество примеров создания полных веб-стеков с применением WebSharper. Например:

- http://www.websharper.com/blogs/2011/6/2059;
- http://www.developerfusion.com/article/124078/building-an-html5-applicationwith-websharper-sitelets-part-1/;
- http://v2matveev.blogspot.com/2010/06/playing-with-websharper.html;
- http://fsharp-code.blogspot.com/2012/07/websharper-slideshow.html.

Они, как и примеры приложений на сайте WebSharper[2], могут пригодиться всем, кто начинает осваивать этот фреймворк.

Пример

При создании приложения To Do на основе WebSharper у нас на выбор имеется несколько вариантов. Несмотря на то, что можно было бы использовать любой из них, я выбрал решение на основе ASP.NET MVC, поскольку этот подход наиболее близок к другим концепциям, обсуждавшимся в этой книге. В дополнение к решению, которое будет создано из шаблона проекта, необходимо также установить NuGet-пакет `WebSharper.JQueryUi`.

Для создания этого примера потребуется несколько классов и модулей. Однако я сосредоточусь только на создании элементов `div` для заданий и реализации механизма буксировки элементов мышью. Эту функциональность реализует следующий фрагмент:

```
namespace Website

open IntelliFactory.WebSharper
```

[1] Начальные сведения о сайтлетах и формлетах на русском языке можно найти по адресу: http://www.gotdotnet.ru/blogs/nesteruk/9289/. – *Прим. перев.*

[2] http://websharper.com/.

```
open IntelliFactory.WebSharper.Html
open IntelliFactory.WebSharper.JQuery
open IntelliFactory.WebSharper.JQueryUI

open Website

module Todo =
    [<JavaScript>]
    let initDrag element =
        let config =
            DraggableConfiguration(cursor = "move", helper = "clone")
        Draggable.New(element, config)

    [<JavaScript>]
    let initDrop (element:Html.Element) =
        let config =
            DroppableConfiguration(
                hoverClass = "ui-state-active", accept = ".draggable")
        let dropZone = Droppable.New(element, config)
        dropZone.OnDrop( fun ev el ->
            JQuery.Of(element.Dom).Append(el.Draggable).Ignore )
        dropZone

    [<JavaScript>]
    let main tasks =
        Div [Attr.Class "droppable"] -< [
            for task in tasks ->
                Div [Attr.Class "ui-widget-content draggable"; Text task]
                |> initDrag ]
        |> initDrop

type IndexControl() =
    inherit Web.Control()

    [<DefaultValue>]
    val mutable Tasks : string list

    [<JavaScript>]
    override x.Body =
        upcast Todo.main x.Tasks
```

Как и в примере с использованием компилятора Pit, здесь применяются атрибуты, чтобы сообщить компилятору WebSharper, какие функции должны быть скомпилированы в код на JavaScript. Первая

функция с таким атрибутом – `initDrag`. Она использует расширение jQuery UI для WebSharper, чтобы обеспечить доступ к jQuery UI API со строгим контролем типов. В данном случае функция принимает элемент и добавляет к нему возможность буксировки мышью.

Функция `initDrop` похожа на функцию `initDrag` и так же использует расширение jQuery UI WebSharper. Как можно предположить из ее имени, данная функция помечает указанный элемент, как область сброса. Она также определяет анонимную функцию, выполняющую некоторые операции в случае сброса элемента в эту область.

Основная задача функции `main` – создать буксируемые элементы. Кроме того она вызывает другие функции, реализующие механизм буксировки мышью. Этот небольшой фрагмент демонстрирует возможности фреймворка WebSharper создания элементов HTML со строгим контролем типов.

В конце фрагмента находится определение типа, наследующего класс `Web.Control`. Этот тип вызывает функцию `main` и запускает в работу весь механизм приложения.

> **Примечание.** Хотя этот пример создан на основе шаблона проекта ASP.NET MVC, тип, наследующий `Web.Control`, точно так же можно встроить в решение на основе другого шаблона, такого как ASP.NET WebForms.

Полные исходные тексты примера можно найти по адресу: https://github.com/dmohl/fs-web-cloud-mobile/tree/master/Ch%205/WebSharperTodo. Как я уже говорил, фреймворк WebSharper обладает гораздо более широкими возможностями, чем использовано в этом примере. Аспекты создания клиентского кода – это лишь вершина айсберга.

В заключение

На протяжении всей книги демонстрировалось, как функциональный стиль программирования может сделать программный код более простым, выразительным и удобочитаемым. Инструменты, представленные в данной главе, позволяют использовать этот стиль и для создания пользовательских интерфейсов. Независимо от того, какой инструмент вы предпочтете –LiveScript, Pit, WebSharper или старый, добрый JavaScript – концепции, описанные в данной главе и книге, откроют перед вами новые горизонты и станут источником вдохновения.

Мы с вами проделали трудный путь. Вы узнали, как на языке F# создавать приложения ASP.NET MVC, службы WCF и HTTP, облачные и мобильные приложения, и многое другое. Теперь у вас есть все необходимое, чтобы приступить к созданию собственных облачных, мобильных и веб-приложений, используя всю мощь языка F#. Кого-то может задаться вопросом: «Куда пойти дальше с этого момента?». Лучший совет, какой я могу дать – идите и создавайте приложения на F#. Чем шире вы будете использовать этот язык, тем больше вы его полюбите!

Приложение А. Полезные инструменты и библиотеки

В этом приложении рассматривается несколько инструментов и библиотек, которые помогут вам в разработке облачных, мобильных и веб-приложений на языке F#.

FAKE (F# Make)

FAKE – это инструмент сборки, похожий на Make и позволяющий автоматизировать последовательность шагов, выполняемых при сборке каждого приложения. Разнообразие операций, которые можно включить в сценарий сборки, бесконечно, но как минимум бывает желательно скомпилировать все проекты и запустить модульные тесты.

Сам инструмент FAKE написан на F# и сценарии, используемые для автоматизации процесса сборки, также пишутся на F#. Я использовал FAKE в таких проектах, как FsUnit, и нахожу его очень полезным! Также я считаю его хорошим способом «протащить» F# в любые группы или организации, которые пока колеблются с вводом F#. Несколько примеров, которые помогут начать использовать FAKE, можно найти по адресу: https://github.com/fsharp/FAKE/tree/develop/Samples.

> **Примечание.** FAKE использует платформу сборки MSBuild, поэтому, вдобавок к богатым возможностям F# и различным расширениям для FAKE, у вас есть возможность использовать все, что предлагает платформа MSBuild.

NuGEt

Много раз в книге я упоминал NuGet – расширение для Visual Studio; однако, я не рассказывал, как им пользоваться. Если NuGet для вас в новинку, лучший способ познакомиться с ним – прочесть

документацию на веб-сайте NuGet[1]. А в следующем разделе представлен более общий обзор.

Основы использования

Существует два основных способа использовать расширение NuGet из среды Visual Studio. Один из них – с помощью консоли диспетчера пакетов. Другой – с помощью диалога диспетчера пакетов. В этой книге я часто предлагал вам установить тот или иной пакет, указывая его имя. Зная имя пакета, его легко можно установить в требуемый проект с помощью консоли диспетчера пакетов

Чтобы отыскать эту консоль, выберите пункт меню **View → Other Windows → Package Manager Console** (Вид → Другие окна → Консоль диспетчера пакетов) или **Tools → Library Package Manager → Package Manager Console** (Сервис → Диспетчер пакетов библиотек → Консоль диспетчера пакетов). В обоих случаях откроется консоль диспетчера пакетов.

После запуска консоли диспетчера пакетов, вы увидите окно, которое выглядит, как показано на рис. А.1.

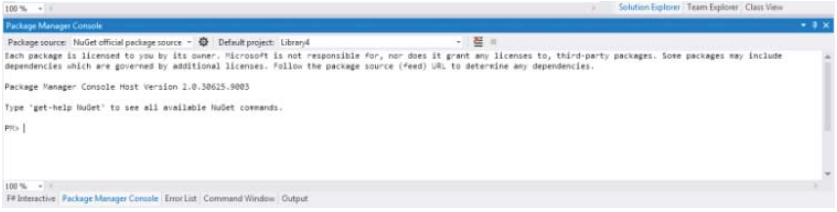

Рис. А.1. Консоль диспетчера пакетов NuGet

Здесь можно выбрать целевой проект, в раскрывающемся списке **Default project** (Проект по умолчанию) и ввести команду, такую как:

```
Install-package packageID
```

чтобы установить пакет `packageID`.

Пользоваться диалогом диспетчера пакетов NuGet так же очень просто. Достаточно щелкнуть правой кнопкой мыши на элементе **References** в требуемом проекте и выбрать пункт контекстного меню **Manage NuGet Packages** (Управление пакетами NuGet...). В от-

[1] http://nuget.org/.

крывшемся диалоге вы сможете найти требуемый пакет и установить его. На рис. А.2 показано, как выглядит окно диалога диспетчера пакетов NuGet.

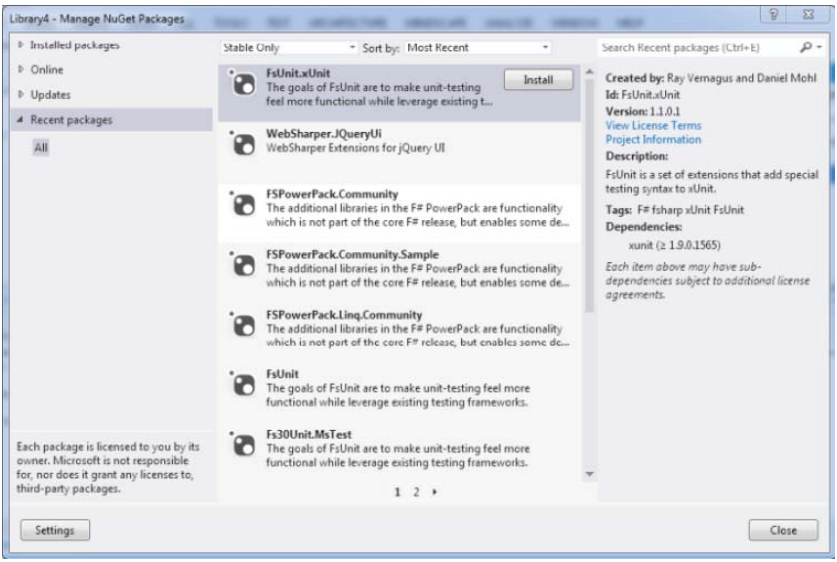

Рис. А.2. Диалог диспетчера пакетов NuGet

Полезные NuGet-пекты

В книге я многократно называл полезные NuGet-пакеты. Следующий список включает ряд других пакетов, которые могут вам пригодиться (описание пакетов взято непосредственно из каталога NuGet):

`PDFsharp` *(автор: PDFsharp-Team)*

PDFsharp – открытая библиотека для платформы .NET, упрощающая создание и обработку документов в формате PDF на любом языке .NET. Одни и те же процедуры рисования могут использоваться для создания документов PDF, вывода графики на экран или печати на любом принтере. Библиотека PDFsharp реализована на основе GDI+.

`FSharpx.Core` *(авторы: mausch, danmohl, sforkmann, panesofglass)*

`FSharpx` – библиотека для платформы .NET, реализующая обобщенные функциональные конструкции поверх базовой биб-

лиотеки языка F#. В первую очередь предназначена для использования в программах на языке F#, но с таким же успехом может использоваться из программного кода на любых других языках, поддерживаемых платформой .NET. В настоящее время реализует:

- несколько стандартных монад: `State`, `Reader`, `Writer`, `Either`, `Continuation`, `Distribution`;
- повторения `Iteratee`;
- проверку приложений и многое другое.

`FSPowerPack.Community` *(автор: danmohl)*

F# PowerPack – это коллекция библиотек и инструментов, разрабатываемая командой F# из Microsoft, для использования в приложениях на языке F#.

`PicoMvc` *(автор: robertpi)*

Тонкая обертка на F# вокруг разных фреймворков для создания легковесных фреймворков MVC.

`FSharp.Testing` *(автор: sforkmann)*

Несколько расширений, упрощающих тестирование кода на F# из тестовых проектов на C#.

`FSharpx.Observable` *(автор: sforkmann)*

`FSharpx` – библиотека для платформы .NET, реализующая обобщенные функциональные конструкции поверх базовой библиотеки языка F#. В первую очередь предназначена для использования в программах на языке F#, но с таким же успехом может использоваться из программного кода на любых других языках, поддерживаемых платформой .NET. Данная библиотека была создана Филом Трелфордом (Phil Trelford) реализует мини-расширения Reactive Extensions (MiniRx) and was authored by Phil Trelford.

`FSharpx.Http` *(автор: sforkmann)*

`FSharpx` – библиотека для платформы .NET, реализующая обобщенные функциональные конструкции поверх базовой библиотеки языка F#. В первую очередь предназначена для использования в программах на языке F#, но с таким же успехом может использоваться из программного кода на любых других языках, поддерживаемых платформой .NET. Данная библиотека предоставляет универсальные функции для работы с HTTP-приложениями.

`Soma` *(автор: taedium)*

`Soma` – фреймворк объектно-реляционного отображения, разработанный на F#. Поддерживает не только F#, но также C# и VB.NET.

`Math.NET Numerics F# Modules` *(авторы: mathnet, cdrnet)*

F# Modules для Math.NET Numerics – математическая основа проекта Math.NET, предоставляющая методы и алгоритмы для решения научных, инженерных и повседневных задач. Компания Numerics образовалась в результате слияния dnAnalytics и Math.NET Iridium.

`FParsec` *(авторы: panesofglass, huwman, Khanage)*

`FApsec` – библиотека парсер-комбинаторов для F#.

`FSharpChart` *(автор: Карл Нолан (Carl Nolan))*

Пакет `FSharpChart` содержит обертки для типов в пространстве имен `System.Windows.Forms.DataVisualization.Charting`[1], упрощающие создание интерактивных диаграмм из F# Interactive.

`Math.NET Numerics F# Module Code Samples` *(авторы: mathnet, cdrnet)*

Этот пакет содержит примеры использования пакета F# Modules для библиотеки Math.NET Numerics. Math.NET Numerics – математическая основа проекта Math.NET, предоставляющая методы и алгоритмы для решения научных, инженерных и повседневных задач.

`FsRavenDbTools` *(автор: robertpi)*

Упрощает использование NoSQL-решения RavenDB из F# (и Newtonsoft.Json).

`FsLinqFixed` *(автор: robertpi)*

Это – библиотека F# `PowerPack.Linq`, основанная на коде F# PowerPack и открытая под лицензией Apache 2.0[2].

`Fracture` *(автор: panesofglass)*

Пакет `Fracture` – реализация сокетов на F# для высокопроизводительных приложений с высокой пропускной способ-

---

[1] http://msdn.microsoft.com/ru-ru/library/system.windows.forms.datavisualization.charting.aspx.

[2] http://fsharppowerpack.codeplex.com/.

ностью. Основан на классе `SocketAsyncEventArgs`, минимизирующем фрагментацию памяти, характерную для шаблона IAsyncResult.

`KinkumaFramework F#` *(F# MVVM Framework; автор: okazuki)*
Фреймворк F# MVVM Framework на основе Prism.

`FSPowerPack.Parallel.Seq.Community` *(автор: danmohl)*
Этот пакет включает в себя `FSPowerPack.Core.Community`, а также библиотеки `Parallel.Seq`. F# PowerPack – это коллекция библиотек и инструментов, разрабатываемая командой F# из Microsoft, для использования в приложениях на языке F#.

`log4f` *(автор: robertpi)*
Тонкая обертка на F# для библиотеки `log4net`.

`FSPowerPack.Community.Sample` *(автор: danmohl)*
Этот пакет целиком включает в себя коллекцию библиотек F# PowerPack, а также простые примеры ее использования. F# PowerPack – это коллекция библиотек и инструментов, разрабатываемая командой F# из Microsoft, для использования в приложениях на языке F#.

`FSharpEnt.Common` *(автор: colinbul)*
Расширения языка F# для корпоративного использования; на данный момент включает `FSharpEnt.Json`, `FSharpEnt.Etl` и `FSharpEnt.Actors`.

`Focco` *(автор: panesofglass)*
`Focco` – простой и быстрый генератор документации, применяемый при использовании стиля «грамотное программирование»[1].

`FSPowerPack.Metadata.Community` *(автор: danmohl)*
Этот пакет включает `FSPowerPack.Core.Community`, а также библиотеки `Metadata`. F# PowerPack – это коллекция библиотек и инструментов, разрабатываемая командой F# из Microsoft, для использования в приложениях на языке F#.

`FSharpEnt.RabbitMq` *(автор: colinbul)*
Обертка для клиента платформы RabbitMq на F#.

---

[1] http://ru.wikipedia.org/wiki/Грамотное_программирование. – *Прим. перев.*

`Samples.WindowsAzure.Marketplace` *(автор: dsyme)*

Пример. Поставщик типов F# 3.0, дающий доступ к наборам данных по адресу: http://datamarket.azure.com/.

`FSharp.Reactive 2.0.120725-rc` *(авторы: sforkmann, panesofglass)*

Обертка на F# для расширения Reactive Extensions.

ExpectThat

ExpectThat – библиотека утверждений, которые могут пригодиться для тестирования сценариев на LiveScript, CoffeeScript и/или JavaScript. На синтаксис ExpectThat большое влияние оказала библиотека FsUnit, поэтому я решил включить ее в это приложение. Как и FsUnit, библиотека ExpectThat поддерживает несколько различных JavaScript-фреймворков тестирования. В настоящее время поддерживаются: QUnit, Jasmine, Mocha и Pavlov. Кроме того, ExpectThat работает и в браузерах, и под управлением Node.js. Исходные тексты библиотеки, а также некоторые примеры можно найти по адресу: https://github.com/dmohl/expectThat.

Простейший способ начать работу с ExpectThat в Visual Studio состоит в следующем:

1. Установить расширение Mindscape Web Workbench (доступно в Visual Studio Gallery)[1].
2. Создать веб-проект из любого понравившегося шаблона.
3. Установить один из NuGet-пакетов, входящих в библиотеку ExpectThat, например, `ExpectThat.Mocha`.
4. Сохранить файл *example.spec.coffee*, чтобы сгенерировался основной JavaScript-файл. Этот файл будет сохранен в папке *Scripts\Specs* веб-проекта, созданного в пункте 2.
5. Открыть *index.html* в браузере.

Код CoffeeScript в файле *example.spec*.coffee показан ниже, а результатом его работы будет список спецификаций, как показано на рис. А.3 (подробнее о CoffeeScript рассказывается в приложении С).

```
describe "Example Mocha Specifications", ->
    foo = "bar"
    describe "When testing should equal", ->
        expectThat -> foo.should equal "bar"
```

[1] http://bit.ly/mindscape-workbench.

```
    describe "When testing shouldnt equal", ->
        expectThat -> foo.shouldnt equal "baz"
    describe "When testing for true", ->
        expectThat -> (foo is "bar").should be true
        expectThat -> (foo is "baz").shouldnt be true
    describe "When testing for false", ->
        expectThat -> (foo is "baz").should be false
        expectThat -> (foo is "bar").shouldnt be false
    describe "When testing to and be", ->
        expectThat -> foo.should be equal to "bar"
        expectThat -> foo.shouldnt be equal to "bah"
    describe "When testing for null or undefined", ->
        testNull = null
        testUndefined = undefined
        expectThat -> (testNull is null).should be true
        expectThat -> (testNull isnt null).shouldnt be true
        expectThat -> (testUndefined is undefined).should be true
        expectThat -> (testUndefined is undefined).shouldnt be false
    describe "When testing for throw", ->
        expectThat -> (-> throw "test exception").should throwException
        expectThat ->
            (-> throw "test exception").should throwException "test
exception"
    describe "When testing for greater than", ->
        expectThat -> 10.should be greaterThan 9
        expectThat -> 9.1.shouldnt be greaterThan 10
    describe "When testing for less than", ->
        expectThat -> 10.should be lessThan 11
        expectThat -> 10.1.shouldnt be lessThan 10
```

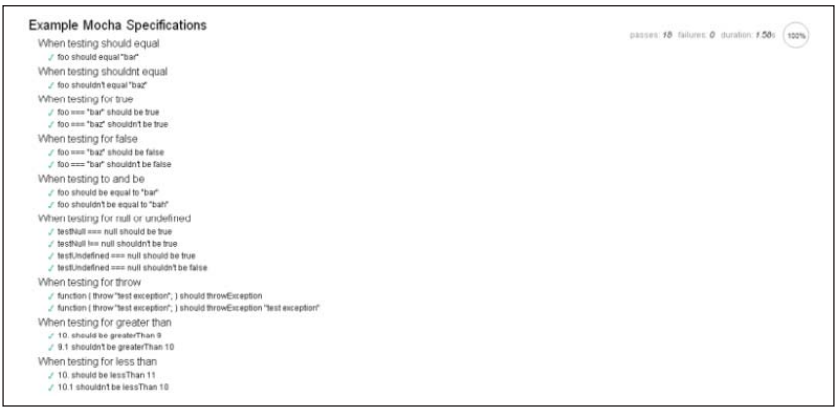

Рис. А.3. Пример использования фреймворка Mocha с помощью ExpectThat

Приложение B. Полезные веб-сайты

В этом приложении перечисляются несколько веб-сайтов, которые могут пригодиться при создании облачных, мобильных и веб-решений. Некоторые из них периодически упоминались в этой книге. Все они являются отличными источниками полезной информации о языке F# и/или о различных библиотеках для веб-разработки.

fssnip.net

На сайте http://fssnip.net можно найти отличную коллекцию примеров на F#. Сайт был создан Томасом Петричеком (Tomas Petricek) с целью дать разработчикам на F# возможность делиться фрагментами исходного кода. На странице **About** (О сайте) Петричек указывает, что fssnip.net «похож на службы Pastebin, но ориентирован исключительно на F#».

В настоящее время сайт содержит множество фрагментов исходного кода? демонстрирующих примеры решения самых разных задач – от создания веб-приложений до игр и предметно-ориентированных языков (Domain Specific Languages, DSL), и их количество постоянно увеличивается. Это делает его отличным ресурсом для тех, что пытается найти пример решения конкретной задачи. Вы тоже можете добавить свой код в эту коллекцию.

tryfsharp.org

На сайте http://tryfsharp.org можно найти отличный учебник для тех, кто начинает осваивать F#. Что более важно, сайт предоставляет веб-версию интерактивной оболочки REPL, позволяющей экспериментировать с языком, ничего не устанавливая и не настраивая. Благодаря ей вы сможете попробовать F# «на вкус», если еще не загрузили компилятор и/или не имеете возможности установить его на свой рабочий компьютер.

Visual Studio Gallery

В книге уже много раз упоминалась галерея Visual Studio Gallery, где можно найти большинство шаблонов и инструментов, о которых я рассказывал. Помимо горстки шаблонов, подробно рассматривавшихся в книге, Visual Studio Gallery содержит массу других полезных расширений для Visual Studio, так или иначе связанных с языком F#.

Самый простой способ найти расширения, связанные с F#, – посетить сайт Visual Studio Gallery[1] и набрать в строке поиска «F#» или «fsharp». Затем вы можете выбрать категорию **Tools** (Инструменты), **Controls** (Элементы управления) или **Templates** (Шаблоны), чтобы сузить круг поиска. Настоятельно рекомендую посмотреть, что там есть, и использовать в своих интересах разработки, созданные сообществом.

jQueryMobile.com

В книге я много раз упоминал библиотеку jQuery Mobile в разных примерах. Если у вас еще не было возможности ознакомиться с ней, настоятельно рекомендую прочитать электронную документацию[2]. Загрузить основные файлы библиотеки, а также исходные тексты инструментов, используемых на сайте jQueryMobile.com, можно по адресу: https://github.com/jquery/jquery-mobile.

---

[1] http://visualstudiogallery.msdn.microsoft.com/.

[2] http://jquerymobile.com/.

Приложение C. Клиентские технологии, совместимые с F#

В последней главе я познакомил вас с некоторыми возможностями, позволяющими реализовать клиентскую часть в функциональном стиле. В этом приложении я представлю некоторые клиентские технологии, совместимые с F#. Это введение не является исчерпывающим описанием этих технологий, но оно поможет вам начать их изучение.

CoffeeScript

Язык CoffeeScript уже упоминался несколько раз в книге. Он похож на LiveScript тем, что код на этом уникальном языке компилируется в код на JavaScript. Как я упоминал в главе 5, это один из прародителей LiveScript, поэтому многое из того, что рассказывалось о LiveScript, относится и к CoffeeScript. Самое большое отличие LiveScript от CoffeeScript в том, что главная цель LiveScript – предоставить возможность разработки в более функциональном стиле.

> **Примечание.** Хотя это приложение рассказывает о клиентских технологиях, следует отметить, что CoffeeScript можно использовать не только на стороне клиента, но и на стороне сервера, благодаря Node.js.

Кто-то может удивиться – зачем выбирать CoffeeScript, когда есть LiveScript. Популярность языка CoffeeScript постоянно растет. Поэтому вокруг него сплотилось многочисленное сообщество, в Интернете можно найти много примеров сценариев на этом языке, кроме того, неуклонно растет количество инструментов поддержки. Самое большее, что вы потеряете, выбрав CoffeeScript, – это несколько функциональных концепций, поддерживаемых языком LiveScript, а также некоторые интересные особенности языка Coco, одного из прародителей LiveScript, о котором рассказывалось в главе 5.

Самый простой способ включить поддержку CoffeScript в Visual Studio – установить расширение Mindscape Web Workbench[1] из Visual Studio Gallery. Это бесплатное расширение с интегрированной поддержкой Visual Studio 2010 и Visual Studio 2012 существенно упрощает создание сценариев на CoffeeScript.

Рассмотрим небольшой пример. Вспомните простой фрагмент на LiveScript, рассматривавшийся в главе 5, который проверяет, приходится ли указанная дата на выходной.

```
weekend = <[ sat sun ]>

isWeekend = (dayOfWeek) ->
     | dayOfWeek in weekend => true
     | _ => false

'sun' |> isWeekend |> console.log
```

Чтобы переписать этот код на CoffeeScript с помощью Mindscape Web Workbench, нужно выполнить следующие шаги:

1. Установить расширение Mindscape Web Workbench (если это еще не сделано).
2. Добавить элемент CoffeeScript в любой выбранный веб-проект и дать ему соответствующее имя.
3. В получившийся файл **.coffee* добавить следующий код:

```
weekend = ["sat", "sun"]

isWeekend = (dayOfWeek) ->
     dayOfWeek in weekend

console.log isWeekend "sun"
```

Вот и все. Расширение Mindscape Web Workbench будет автоматически скомпилирует файл **.coffee* и сохранит результат в файл **.js* с тем же именем. После этого можно будет добавить ссылку на файл **.js* в веб-приложение, как на любой другой файл **.js*.

Sass

Sass – инструмент, относящийся к категории препроцессорных языков и предназначенный для создания каскадных таблиц стилей

[1] http://visualstudiogallery.msdn.microsoft.com/2b96d16a-c986-4501-8f97-8008f9db141a.

(Cascading Style Sheets, CSS). Препроцессорные языки CSS позволяют легко писать удобные и гибкие определения стилей CSS, используя более выразительный синтаксис, которые затем конвертируются в старый добрый формат CSS. Кроме того, что определения таблиц стилей CSS становятся более понятными и простыми в сопровождении; еще одна причина использовать Sass – возможность заменять фигурные скобки значимыми пробелами.

> **Примечание.** Sass – это не единственный препроцессорный язык для определения таблиц стилей CSS. В числе других популярных альтернатив можно назвать LESS и Stylus. Для экономии места я остановлюсь только на Sass; однако рекомендую познакомиться и с другими альтернативами, чтобы решить, какая из них лучше отвечает вашим требованиям.

Посмотрим, как действует Sass. В примере приложения To Do на LiveScript (я его описывал в главе 5), для добавления стилей использовался файл *Site.css*[1]. Этот файл содержит следующий код:

```
.detail {
     margin: 10px 0px;
}

.column-header {
     padding-bottom: 5px;
     margin: 5px 0;
     border-bottom: 1px solid #EEE;
}

.draggable {
     height: 50px;
     -webkit-border-radius: 4px;
     -moz-border-radius: 4px;
     border-radius: 4px;
     -webkit-box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.05);
     -moz-box-shadow: inset 0 1px 1px rgba(0,0,0,0.05);
     box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.05);
     margin-bottom: 10px;
     padding: 10px;
     vertical-align:middle;
}
```

[1] http://bit.ly/SYCZrP.

```
.tasks {
     min-height: 100px;
}
```

Это довольно типичный CSS-файл, но было бы хорошо, если можно было бы убрать повторяющиеся элементы стилей, связанные с поддержкой разных браузеров, а также символы, которые не добавляют удобочитаемости, такие как фигурные скобки и точки с запятой. И то, и другое, и даже больше, можно сделать с помощью Sass. Следующий фрагмент решает эти проблемы:

```
@mixin border-radius($pxl)
     -webkit-border-radius: $pxl
     -moz-border-radius: $pxl
     border-radius: $pxl

@mixin box-shadow($val)
     -webkit-box-shadow: $val
     -moz-box-shadow: $val
     box-shadow: $val

.detail
     margin: 10px 0px

.column-header
     padding-bottom: 5px
     margin: 5px 0
     border-bottom: 1px solid #EEE

.draggable
     height: 50px
     @include border-radius(4px)
     @include box-shadow(inset 0 1px 1px rgba(0, 0, 0, 0.05))
     margin-bottom: 10px
     padding: 10px
     vertical-align: middle

.tasks
     min-height: 100px
```

Это пример демонстрирует лишь малую часть возможностей Sass. Помимо них имеется также возможность создавать и использовать переменные, включать операторы и функции в определения CSS,

и пользоваться такими приятными мелочами, как интерполяция строк. Более подробную информацию можно получить на веб-сайте Sass[1].

Underscore.js

В главе 5, я мельком отметил библиотеку `prelude.ls`, которая поддерживает ряд функций, позволяющих писать клиентский код для веб-приложений в более функциональном стиле. Библиотека `Underscore.js` была написана создателем CoffeeScript и преследует те же цели, что и `prelude.ls`, но имеет более широкие возможности (не все из которых являются характерными для функционального стиля).

Сайт проекта Undercode.js[2] содержит большое количество примеров, поэтому я перечислю только основные функции:

- функции для работы с коллекциями: `map`, `reduce`, `filter`, `all`, `any`, `sortBy` и `groupBy`;
- функции для работы с массивами: `first`, `last`, `union`, `intersect`, `flatten` и `zip`;
- вспомогательные функции: `bind`, `memoize`, `throttle` и `once`;
- функции для работы с объектами: `isString`, `isDate`, `isBoolean`, `has`, `clone` и `tap`;
- прочие особенности, включая механизм шаблонов и возможность составления цепочек вызовов.

Примечание. Это далеко не полный список функций и возможностей, поддерживаемых библиотекой `Underscore.js`. Я настоятельно рекомендую посетить сайт проекта Underscore.js, чтобы получить более полную информацию.

---

[1] http://sass-lang.com/.

[2] http://underscorejs.org/.

Об авторе

Дэниел Мол (Daniel Mohl) – наиболее ценный специалист в команде разработчиков F# корпорации Microsoft, опытный разработчик приложений на языках F# и C#, блоггер, оратор и организатор мероприятий. Его блог находится по адресу: http://blog.danielmohl.com/, и вы можете последовать за ним в Твиттере: https://twitter.com/dmohl.

Предметный указатель

(символ хеша), комментарии
в LiveScript, 172
.NET 4.5 и IIS 8, пример использования веб-сокетов, 139
`` (двойные обратные апострофы), 97
{m}brace, фреймворк, 134
|>, прямой конвейерный оператор, 26, 169
применение для композиции функций, 36
<|, обратный конвейерный оператор, 169
<<, оператор обратной композиции функций, 169
>>, оператор прямой композиции функций, 169

Active Directory Federation Services (ADFS), 120
ApiController, тип, 76
ASP.NET MVC 4, фреймворк
создание приложений на F#, 17
создание более функционального контроллера, 38
взаимодействие с базами данных, 28
использование преимуществ F#, 34
кеширование с применением MailboxProcessor, 46
конвейеры и частичное применение функций, 36
контроллеры и модели, 24
переход на функциональную парадигму, 34
создание собственных вычислительных выражений, 58
стиль продолжений, 57
улучшение времени отклика с помощью асинхронных операций, 44
упрощение за счет сопоставления с образцом, 40
шаблоны проектов, 18
шины сообщений, 51
Async.AwaitTask, метод, 68
AsyncBuilder, тип, 92
Async.StartWithContinuations, метод, 81
AutoOpen, атрибут, 161

BinaryMessageFormatter, класс, 52

C#, универсальные репозитории, 35
CacheAgent, использование, 49
ChannelFactory, 69
CLIMutable, атрибут, 65, 160
Cloud Numerics, библиотека, 135
Coco, 173
CoffeeScript, 173, 196
CompositeType, тип, 65
Contracts, проект, 63
CouchDB, 163

D3, библиотека, 140
DataServiceKey, атрибут, 114
DbContext, класс, 29
Dispose, метод, контроль над вызовом с помощью функции using, 146
DomEntryPoint, атрибут, 178
DynamicDictionary, тип, 88

each, функция из библиотеки prelude.ls, 172
EndPointAddress, тип, 67
Entity Framework, фреймворк, взаимодействие с базами данных SQL Server из ASP.NET MVC, 28
ExpectThat, библиотека утверждений, 192

F#
Pit, 173
использование Frank, 90
использование Nancy, 87
использование Service Stack, 84
использование записей, 72
облачные приложения, 105

Fog, библиотека для взаимодействий с Windows Azure из F#, 109
авторизация на основе заявок в Windows Azure, 122
аутентификация и авторизация в Windows Azure, 119
аутентификация на основе заявок в Windows Azure, 121
взаимодействие с хранилищами данных в Windows Azure, 110
использование преимуществ Windows Azure Service Bus, 116
очереди Windows Azure, 116
служба SQL Azure в Windows Azure, 115
служба хранения очередей в Windows Azure, 114
создание веб-роли, 124
создание и развертывание приложений F# на платформе Azure, 106
создание масштабируемых приложений Windows Azure, 123
создание рабочей роли, 108
темы Windows Azure, 117
создание веб-приложений на основе ASP.NET MVC 4
создание более функционального контроллера, 38
использование преимуществ F#, 34
кеширование с применением MailboxProcessor, 46
конвейеры и частичное применение функций, 36
переход на функциональную парадигму, 34
создание собственных вычислительных выражений, 58
стиль продолжений, 57
улучшение времени отклика с помощью асинхронных операций, 44
упрощение за счет сопоставления с образцом, 40
шины сообщений, 51
тестирование приложений, 94
установка шаблона проекта WCF, 63
FAKE, инструмент сборки, 186
F#/C# ASP.NET Web API, шаблон проекта, 74
Fleck, библиотека, создание сервера веб-сокетов, 144
Fluent Security, библиотека, 123
Fog, библиотека, 109, 124
взаимодействие с хранилищами данных в Windows Azure, 110
Frank, фреймворк веб-служб, 90
добавление перенаправления привязки, 93
FSharp.Core, сборка, версии, 93
FSharp.Data.TypeProviders, сборка, 32, 67
FSharpx.TypeProviders, библиотека, 82
FSharpx, коллекция библиотек, 82
FSPowerPack.Linq.Community, пакет NuGet, 29
fssnip.net, веб-сайт, 194
FsUnit, фреймворк, 99

Global.fs, файл, ASP.NET Web API, служба, 75

Hadoop, фреймворк, 136
HandleErrorAttribute, тип, 75
HttpClient, использование с веб-службой ASP.NET Web API, 79
HttpMessageHandlers, тип, 84
HttpResource, тип, 92
Hub, класс (SignalR), 150
клиентская сторона, 152
серверная сторона, 151

IDisposable, интерфейс, классы, использование ключевого слова new, 112
IHttpHandler, интерфейс, 140
IIS 8 и .NET 4.5, пример использования веб-сокетов, 139
ImpromptuInterface.FSharp, библиотека, 152
IService1, интерфейс, 66
iTunes, HTTP-служба, 83

JavaScript
D3, библиотека, 140
и LiveScript, 169
клиент для сервера SignalR, 149
код в приложениях Windows Azure, 124
основной язык создания клиентского кода веб-приложений, 166
подключение к серверу веб-сокетов, 143
jQuery
getJSON, функция, 79
поддержка Pit, 176

jQuery.each, функция, 173
jQueryMobile.com, веб-сайт, 195
jQuery Mobile, библиотека, 124, 153
jQuery UI
  поддержка Pit, 176
  расширение для WebSharper, 184
JsObject, атрибут, 178
JSON
  jQuery getJSON, функция, 79
  поставщик типов, 82
Json.NET, библиотека, 79
JsonServiceClient, тип, 86
Js, атрибут, 178

let, ключевое слово вместо use, 29
LINQ, 31
LiveScript, 168
  и CoffeeScript, 196
  и JavaScript, 169
  преимущества, 168
  применение, 169
  пример приложения, 170
Lucene, библиотека, 162

Make, 186
MapHttpRouteSettings, тип, 75
MessageQueue, объект, создание, 52
Microsoft.VisualStudio.QualityTools.
UnitTestFramework, сборка, 96
Microsoft.WebSockets, NuGet-пакет, 140
Mindscape Web Workbench, 197
MongoDB, 159
MongoFs, пакет, 159
MSBuild, 186
MSTest, шаблон проекта, 96
MVVM (Model-View-ViewModel – Модель-Представление-Модель-Представления), шаблон проектирования, 157

Nancy, фреймворк веб-служб, 87
NaturalSpec, 102
new, ключевое слово, использование с классами, реализующими IDisposable, 112
NoSQL, хранилища данных, 117
  объединение с F#, 158
    CouchDB, 163
    MongoDB, 159
    RavenDB, 162
NuGet, расширение, 186
  основы использования, 187
  полезные пакеты, 188

ObjectId, тип, 160
Option, тип, 43

Pit, 173
  преимущества, 174
  применение, 175
  пример приложения To Do, 176

RavenDB, 162
Response, объект, преобразование строк, 88
REST API (WCF), 73
RestServiceBase, тип, 85

Sass, 197
Seq.filter, функция, 123
Seq.fold, функция, 123
Seq.head, функция, 123
Seq.map, функция, 123
Service Stack, фреймворк веб-служб, 84
SignalR.Hosting.Self, пакет, 151
SignalR, библиотека
  клиент на F#, 150
  создание хаба, 150
    клиентская сторона, 152
    серверная сторона, 151
SignalR, библиотеки, 147
  клиент на JavaScript, 149
  создание постоянного соединения, 148
SimpleBus, библиотека, 52
SQL Azure, 126, 128
SqlDataConnection, поставщик типов, 32, 96
SQL Express, 107
SQL Server, взаимодействие с базами данных из ASP.NET MVC, 28
SupportedProducts, элемент, в файле extension.vsixmanifest, 175
System.Json, библиотека, 79
System.Runtime.Serialization, сборка, 69
System.ServiceModel, сборка, 67

try/close/catch/abort, шаблон реализации WCF-клиента, 70
tryfsharp.org, веб-сайт, 194

Underscore.js, 200
Unquote, 101
using, функция, 146
Utils.NullCheck, функция, 43

Visual Studio, поддержка Pit, 175
Visual Studio Gallery
  веб-сайт, 195
  поиск шаблона проекта F# WCF, 63
VoteCounts, запись, 141

WebSharper, 179
  преимущества, 180
  приложения To Do, 182
WebSocketChartHandler, класс, 140
WebSocketHandler, класс, 142
Web Sockets, 138
Windows Azure, 105
  Fog, библиотека для взаимодействий с Windows Azure из F#, 109
  авторизация на основе заявок, 122
  аутентификация и авторизация, 119
  аутентификация на основе заявок, 121
  взаимодействие с хранилищами данных в Windows Azure, 110
  использование преимуществ Windows Azure Service Bus, 116
  очереди, 116
  служба SQL Azure, 115
  служба хранения очередей, 114
  служба хранения таблиц, 112
  создание веб-роли, 124
  создание и развертывание приложений F# на платформе Azure, 106
  создание масштабируемых приложений, 123
  создание рабочей роли, 108
  темы, 117
Windows Azure Caching, 132
Windows Azure SDK для .NET
  загрузка и установка, 106
  пошаговое руководство, 106
Windows Azure Service Bus, 116
WsdlService, поставщик типов, 67
XmlMessageFormatter, класс, 52

Абстрактное синтаксическое дерево и ReflectedDefinitionAttribute, 178
Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST), 29
Автоматическое масштабирование, 133
Автономные службы (self-host services), 77
Авторизация на основе заявок, 122
Асинхронные потоки операций, 44
Аутентификация и авторизация с применением службы управлением доступом (ACS), 120
Аутентификация на основе заявок, 121

Библиотеки, 186

Веб-проект, 63
Веб-сайты, полезные, 194
Веб-службы, создание на F#, 62
  использование Frank, 90
  использование Nancy, 87
  использование Service Stack, 84
  исследование получившегося проекта, 64
  создание службы ASP.NET Web API, 73
    взаимодействие с HTTP-службой, 78
    шаблон проекта, 74
  тестирование приложений, 94
    FsUnit, фреймворк, 99
    NaturalSpec, 102
    Unquote, 101
    подготовка, 94
    улучшение тестов с применением F#, 97
  установка шаблона проекта WCF, 64
Веб-сокеты, масштабирование, 138
  .NET 4.5 и IIS 8, пример использования веб-сокетов, 139
  создание сервера веб-сокетов с помощью Fleck, 144
Выражения объектов (object expressions), 98
Высшего порядка, функции, 35
Вычислительные выражения, 30
  создание собственных вычислительных выражений, 58

Двойные обратные апострофы (``), 97
Делегирования, шаблон, 84

Записи преимущества перед классами F#, 64
Запросов, синтаксис, 30, 31
  с применением поставщиков типов, 32

Инверсия управления, 30
Инструменты и библиотеки, 186
Информация о маршрутизации ASP.NET Web API, служба, 76

Каррированные функции, 37, 169
Кеширование, 131
  мемоизация, 115
  распределенное, 131
  с применением MailboxProcessor, 46
Классы и записи, 72
Клиент для сервера SignalR, 149
Клиентские технологии, совместимые с F#, 196
Композиция функций, 36, 169
Конвейеры, 36
Контроллеры
  ASP.NET Web API, 124
    тестирование, 94
  создание более функционального контроллера, 38
  перегрузка конструктора для контроллера на основе ASP.NET MVC, 30
  приложений ASP.NET Web API, 75
  сопоставление с образцом, 42

Масштабируемость приложений на платформе Windows Azure, 123
Масштабируемые мобильные и веб-приложения, создание, 137
  .NET 4.5 и IIS 8, пример использования веб-сокетов, 139
  использование SignalR
    клиент на F#, 150
    клиент на JavaScript, 149
    создание хаба, 150
  использование библиотек SignalR, 147
  масштабирование с применением веб-сокетов, 138
  обретаем мобильность, 153
    добавляем поддержку Windows Phone 7, 155
    с помощью библиотеки jQuery Mobile, 153
  объединение F# и NoSQL, 158
    CouchDB, 163
    MongoDB, 159
    RavenDB, 162
  постоянное соединение с использованием SignalR, 148
  создание сервера веб-сокетов с помощью Fleck, 144
Мемоизация, 115
Мобильные приложения, 153
  на основе библиотеки jQuery Mobile, 153
Модель асинхронного программирования (Asynchronous Programming Model, APM), 68
Модель-Представление-Модель-Представления (Model-View-ViewModel, MVVM), шаблон проектирования, 157

Неизменяемость записей, 72

Облачные приложения на F#, 105
  Fog, библиотека для взаимодействий с Windows Azure из F#, 109
  авторизация на основе заявок в Windows Azure, 122
  аутентификация и авторизация в Windows Azure, 119
  аутентификация на основе заявок в Windows Azure, 121
  взаимодействие с хранилищами данных в Windows Azure, 110
  использование преимуществ Windows Azure Service Bus, 116
  очереди Windows Azure, 116
  служба SQL Azure в Windows Azure, 115
  служба хранения очередей в Windows Azure, 114
  служба хранения таблиц в Windows Azure, 112
  создание веб-роли, 124
  создание и развертывание приложений F# на платформе Azure, 106
  создание масштабируемых приложений Windows Azure, 123
  создание рабочей роли, 108
  темы Windows Azure, 117
Объединение F# и NoSQL, 158
  CouchDB, 163
  MongoDB, 159
  RavenDB, 162
Объекты передачи данных (Data Transfer Object, DTO), 85

Перенаправление привязки
  добавление в веб-службу на основе Frank, 93
Поставщики типов, 32
  JSON, 82
  SqlDataConnection, 96
  WsdlService, 67

Постоянное соединение, с применением SignalR, 148
Препроцессорные языки CSS, 197

Рабочие роли (F#), создание, 108
Разделение ответственности на команды и запросы (Command Query Responsibility Segregation, CQRS), 54
Размеченные объединения, 47
Разработка интерфейсов, в функциональном стиле, 166
  LiveScript, 168
  Pit, 173
  WebSharper, 179
  пример приложения To Do, 167
Расширение типов (type extensions), 88
Репозитории
  применение функций вместо объектов, 37
  создание класса репозитория, 29
  универсальный модуль Repository, 35

Сайтлеты (sitelets), 181
Сети доставки содержимого (Content Delivery Network, CDN), 133
Симуляторы и эмуляторы для отладки мобильных приложений, 154
Служба SQL Azure в Windows Azure, 115
Служба управления доступом (Access Control Service, ACS), 120
Служба хранения больших двоичных объектов, Windows Azure, 110
Служба хранения очередей в Windows Azure, 114
Служба хранения таблиц в Windows Azure, 112
Создание веб-служб, 62
  использование службы, 67
  исследование получившегося проекта, 64
  создание службы ASP.NET Web API, 73
Создание службы ASP.NET Web API, 73
  взаимодействие с HTTP-службой, 78
  шаблон проекта, 74
Сообщения
  SimpleBus, библиотека (пример), 52
  использование шин сообщений, 51
  как значения типа размеченного объединения, 47
Сопоставление с образцом, 40, 169
  для проверки типа значения, 112
Стиль продолжений (Continuation-Passing Style, CPS), 57
Строка подключения веб-приложения на C#, 31
Строки подключения
  хранилища больших двоичных объектов в Windoes Azure, 111
  хранилище таблиц в Windows Azure, 113
Структуры и записи, 72

Тестирование
  веб-служб, 94
    FsUnit, фреймворк, 99
    NaturalSpec, 102
    Unquote, 101
    подготовка, 94
    улучшение тестов с применением F#, 97
  и неизменяемые записи, 73

Формлеты (formlets), 182

Цитируемые выражения, 29
Цитируемые выражения (quoted expressions), использование для статической проверки, 101

Частичное применение функций, 36

Шаблон асинхронного программирования на основе заданий (Task-based Asynchronous Pattern), 68
Шаблоны проектов
  MSTest, 96
  Sample Web Application (Sitelets), 182
  Windows Azure, 107
  установка шаблона проекта WCF, 63
Шины сообщений, 51

Эмуляторы и симуляторы для отладки мобильных приложений, 154

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по поч товому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Дэниел Мол

Создание облачных, мобильных и веб-приложений на F#

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод с анлийского *Киселев А. Н.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 05.01.2013. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 13. Тираж 200 экз.

Веб-сайт издательства: www.dmk-press.ru