Функциональное программирование

Дэниел Мол

Создание облачных, мобильных и веб-приложений на F#

ДМК Пресс
издательство

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

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.

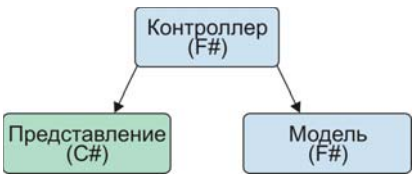

Рис. 1.1. Шаблон проектирования MVC с обозначением типов компонентов

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**.

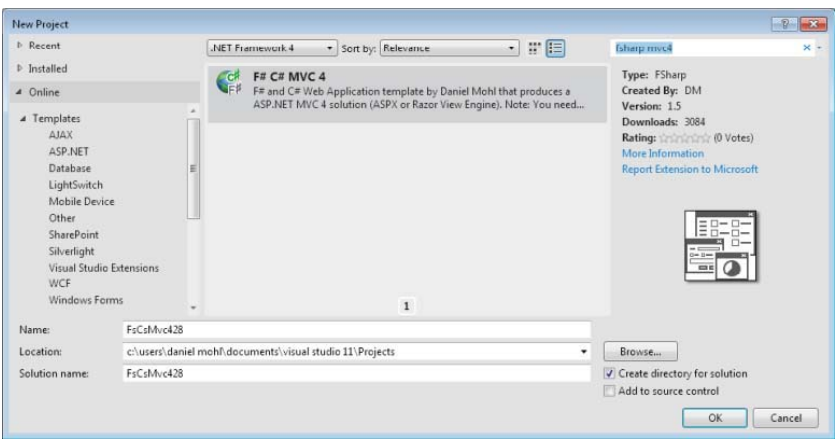

Рис. 1.2. Поиск шаблона проекта в галерее Visual Studio Gallery

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