



From Technologies to Solutions

ASP.NET MVC 1.0 Quickly

Design, develop, and test powerful and robust web applications the agile way, with MVC framework

Maarten Balliauw

[PACKT]
PUBLISHING

ASP.NET MVC 1.0 Quickly

Design, develop, and test powerful and robust web applications the agile way, with MVC framework

Maarten Balliauw



ASP.NET MVC 1.0 Quickly

Copyright © 2009 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2009

Production Reference: 1100309

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-54-2

www.packtpub.com

Cover Image by Maarten Balliauw (maarten@maartenballiauw.be)

Credits

Author

Maarten Balliauw

Production Editorial Manager

Abhijeet Deobhakta

Reviewers

Jerry Spohn

Joydip Kanjilal

Editorial Team Leader

Akshara Aware

Senior Acquisition Editor

Douglas Paterson

Project Team Leader

Lata Basantani

Development Editor

Shilpa Dube

Project Coordinator

Joel Goveya

Technical Editor

Aanchal Kumar

Proofreader

Dirk Manuel

Copy Editor

Sumathi Sridhar

Production Coordinator

Shantanu Zagde

Indexer

Monica Ajmera

Cover Work

Shantanu Zagde

About the author

Maarten Balliauw has a Bachelor's Degree in Software Engineering and has about eight years of experience in software development. He started his career during his studies where he founded a company that did web development in PHP and ASP.NET. After graduation, he sold his shares and joined one of the largest ICT companies in Belgium, RealDolmen, where he continued web application development in ASP.NET and application life cycle management in Visual Studio Team System. He is a Microsoft Certified Technology Specialist in ASP.NET and works with the latest Microsoft technologies such as LINQ and ASP.NET 3.5. He has published many articles in both .NET and PHP literature, such as *MSDN* magazine Belgium, and *PHP Architect*. Ever since the first announcement of the ASP.NET MVC framework, he has been following all its developments and features.

Blog: <http://blog.maartenballiauw.be>

E-mail: maarten@maartenballiauw.be

First of all, I would like to thank the people at Packt Publishing for their guidance on writing this book. This has really been helpful and much appreciated! I also wish to thank all of the reviewers for their efforts in getting all of the ASP.NET MVC community tech preview quirks out in the end.

Saving the best for last, I want to thank my family and my wife for their patience, even when it meant sacrificing valuable holiday and weekend time due to publication deadlines.

About the reviewers

Jerry Spohn is a Manager of Development for a medium-sized software development firm in Exton, Pennsylvania. His responsibilities include managing a team of developers and assisting in architecting a large multi-lingual, multi-currency loan account system, written in COBOL and JAVA. He is also responsible for maintaining and tracking a system-wide program and database documentation web site that uses DotNetNuke as the portal for this information.

Jerry is the owner of Spohn Software LLC – a small consulting firm that helps small businesses in the area with all aspects of maintaining and improving their business processes. This includes helping them with the creation and maintenance of web sites, general office productivity issues, and computer purchasing and networking. Spohn Software, as a firm, prefers to teach their clients how to solve their problems internally, rather than require a long-term contract, thereby making the business more productive and profitable in the future.

Jerry currently works and resides in Pennsylvania, with his wife, Jacqueline, and his two sons, Nicholas and Nolan.

Joydip Kanjilal is a Microsoft Most Valuable Professional in ASP.NET. He has over 12 years of industry experience in IT, including more than six years with Microsoft .NET and its related technologies. He was selected as the MSDN Featured Developer of the Fortnight (MSDN) and was also selected as the Community Credit Winner at www.community-credit.com several times. Joydip has authored the following books:

- *Entity Framework Tutorial* (Packt Publishing)
- *Pro Sync Framework* (APRESS)
- *Sams Teach Yourself ASP.NET Ajax in 24 Hours* (Sams Publishing)
- *ASP.NET Data Presentation Controls Essentials* (Packt Publishing)

Joydip has authored more than 150 articles for some of the most technology reputable sites such as, www.asptoday.com, www.devx.com, www.aspalliance.com, www.aspnetpro.com, www.sql-server-performance.com, www.sswug.com, and so on. A lot of these articles have been selected for inclusion on www.asp.net — Microsoft's official site for ASP.NET. Joydip was also a community credit winner at www.community-credit.com a number of times.

Joydip is currently working as a Lead Architect in a reputed company in Hyderabad, India. He has several years of experience in designing and architecting solutions for various domains. His technical strengths include C, C++, VC++, Java, C#, Microsoft .NET, Ajax, Design Patterns, SQL Server, Operating Systems, and Computer Architecture.

Joydip blogs at: <http://aspadvice.com/blogs/joydip> and spends most of his time writing books and articles. When not at work, Joydip spends time with his family, playing chess, and watching cricket and soccer.

Table of Contents

Preface	1
Chapter 1: ASP.NET MVC	7
Model-view-controller	7
View	8
Controller	8
The ASP.NET MVC framework	9
Driving goals of the ASP.NET MVC framework	10
Comparing ASP.NET MVC and ASP.NET Webforms	11
Choosing between ASP.NET MVC and ASP.NET Webforms	13
Summary	14
Chapter 2: Your First ASP.NET MVC Application	15
Creating a new ASP.NET MVC web application project	16
What's inside the box?	18
Strong-typed ViewData	24
Creating a new view	26
Unit testing the controller	27
Summary	29
Chapter 3: Handling Interactions	31
Creating a form	33
Creating a form using HTML	33
Creating a form using HtmlHelper	34
Handling posts	37
Request variables	37
Updating objects from request variables	37
Action method parameters	38
Handling file uploads	39
Creating an upload form	39
Creating an upload controller action	39

Using the ModelBinder attribute	40
Using the default ModelBinder	41
Creating a custom ModelBinder	43
Validating data	45
Summary	49
Chapter 4: Components in the ASP.NET MVC framework	51
The ASP.NET MVC request life cycle	51
The RouteTable is created	52
The UrlRoutingModule intercepts the request	53
The routing engine determines the route	53
The route handler creates the associated IHttpHandler	53
The IHttpHandler determines the controller	53
The controller executes	54
A ViewEngine is created	54
The view is rendered	54
Extensibility	54
Route objects	54
MvcRouteHandler	55
ControllerFactory	55
Controller	55
ViewEngine	55
View	56
The model in depth	56
Creating a model	56
Enabling validation on the model	58
The controller in depth	60
Creating a controller	60
Rendering data on the response	61
Reading data from the request	62
Action method selection	63
Handling unknown controller actions	64
Action method attributes	66
The view in depth	68
Location of views	69
Creating a view	70
Master pages	71
View markup	72
Partial views	74
Action filters	75
IAuthorizationFilter	75

IActionFilter	76
IResultFilter	76
IExceptionFilter	76
Summary	77
Chapter 5: Routing	79
What is ASP.NET routing?	79
ASP.NET routing versus URL rewriting	80
UrlRoutingModule	80
Route patterns	81
Defining routes	82
Parameter constraints	84
Catch-all routes	85
Routing namespaces	86
Combining ASP.NET MVC and ASP.NET in one web application	88
Creating URLs from routes	89
Summary	90
Chapter 6: Customizing and Extending the ASP.NET MVC Framework	91
Creating a control	92
Creating a filter attribute	96
Creating a custom ActionResult	101
Creating a ViewEngine	105
Summary	113
Chapter 7: Using Existing ASP.NET Features	115
Session State	116
Reading and writing session data	116
Configuring session state	117
TempData	119
Membership, authentication, and authorization	120
Configuring web site security	121
Implementing user and role based security in a controller	122
Configurable authentication options	125
Caching	127
Globalization	129
Resources	129
Using local resources	130
Using global resources	132
Setting language and culture preferences	132
Mixing ASP.NET Webforms and ASP.NET MVC	135
Plugging ASP.NET MVC into an existing ASP.NET application	135

Plugging ASP.NET into an existing ASP.NET MVC application	139
Sharing data between ASP.NET and ASP.NET MVC	140
Building views at compile time	142
Summary	143
Chapter 8: AJAX and ASP.NET MVC	145
Different AJAX frameworks	145
XMLHttpRequest	146
JavaScript Object Notation (JSON)	146
ASP.NET AJAX	147
ASP.NET MVC AJAX helper	147
Working with JsonResult	150
jQuery	152
jQuery syntax	153
Using jQuery with ASP.NET MVC	154
Working with JsonResult	157
Using jQuery UI	159
Summary	164
Chapter 9: Testing an Application	165
Unit testing	166
Unit testing frameworks	166
Hello, unit testing!	166
Generating unit tests	168
Testing the action method	170
Mocking frameworks	172
Testing the Login action method	174
Mocking ASP.NET components	176
Testing routes	178
Testing UpdateModel scenarios	179
Summary	182
Chapter 10: Hosting and Deployment	183
Platforms that can be used	183
Differences between IIS 7.0 integrated and classic mode	184
Hosting an ASP.NET MVC web application	186
Creating a wildcard script map in IIS 7.0	187
Creating a wildcard script map in IIS 6.0	188
Modifying the route table to use file extensions	189
Summary	191
Appendix A: Reference Application— CarTrackr	193
CarTrackr functionality	193
Home page	194

Login screen	195
List of cars	196
Car details	197
Refuellings list	198
Data layer	199
Linq to SQL model	200
Repository pattern	201
Dependency injection	202
How CarTrackr controllers are built	203
Using Unity for dependency injection	204
ASP.NET MVC Membership Starter Kit	207
Form validation	208
ASP.NET provider model	210
Unit testing CarTrackr	212
Unit tests in CarTrackr	213
Mock repository	214
Summary	215
Appendix B: ASP.NET MVC Mock Helpers	217
RhinoMocks	217
Moq	220
TypeMock	222
Appendix C: Useful Links and Open Source Projects	
Providing Additional Features	225
Information portals	225
ASP.NET/MVC	225
Aspdotnetmvc.com	226
DotNetKicks.com: Articles tagged with ASP.NET MVC	227
Blogs	227
Open source projects providing additional features for the ASP.NET MVC framework	228
ASP.NET MVC Design Gallery	228
MVC Contrib	228
xVal validation framework	228
ASP.NET MVC Membership Starter Kit	229
XForms	229
jQuery for ASP.NET MVC	229
Simple ASP.NET MVC controls	229
Alternative view engines	229
Spark view engine	229
NHaml view engine	230

Table of Contents

NVelocity view engine	230
Example ASP.NET MVC applications	230
MVC storefront	230
FlickrExplorer	230
Yonkly	231
Kigg	231
CarTrackr	231
Index	233

Preface

Over the years, people have been asking the ASP.NET support team for the ability to develop web applications using a **model-view-controller (MVC)** architecture. In October 2007, Scott Guthrie presented the first preview of the ASP.NET MVC framework. Ever since, interest in this product has been growing, and many example applications and components have been released on the Internet by enthusiastic bloggers and Microsoft employees.

ASP.NET MVC 1.0 Quickly was written to help people who have a basic knowledge of ASP.NET Webforms to quickly get up-to-speed with developing ASP.NET MVC applications. The book starts by explaining the MVC design pattern, and follows this with a bird's eye-view of what the ASP.NET MVC framework has to offer. After that, each chapter focuses on one aspect of the framework, providing in-depth details of the components that comprise the ASP.NET MVC framework. For each of the concepts explained, a to-the-point example application is provided, demonstrating the theory behind the concept.

By the time you finish this book, you'll be well on your way to mastering the ASP.NET MVC framework, and will have the confidence to build your own ASP.NET MVC applications.

What this book covers

Chapter 1 describes the MVC software design pattern, and how it can be used in application architecture. We also look at the reason why Microsoft started the ASP.NET framework project, and how it compares with ASP.NET Webforms.

Chapter 2 describes the ASP.NET MVC project template that is installed in Visual Studio. A simple application is built, briefly touching on all of the aspects of the ASP.NET MVC framework.

Chapter 3 describes how interactions with the model are handled through a request/response scenario. A simple application where data is displayed and posted to the web server is built, to demonstrate the concepts described.

Chapter 4 takes us through the components that comprise the ASP.NET MVC framework, covering the request lifecycle and all of the components, including model, view and controller, in depth. You will also take a look at some useful concepts such as action filters and the validation of data.

Chapter 5 describes what ASP.NET routing is, and how it works. We will also take a look at how a URL is transformed into a call to an ASP.NET MVC controller. Next, this chapter shows you how an ASP.NET MVC application can be combined with an ASP.NET Webforms application.

Chapter 6 describes how you can customize and extend the ASP.NET MVC framework. You will learn how to build a control, or so-called partial view, how to create an action filter, and how to create a custom ActionResult. You will even build your own view engine that supports simple HTML markup, completely replacing ASP.NET MVC's default view engine.

Chapter 7 describes how you can use existing ASP.NET features, including master pages, sessions, membership, and internationalization, in the ASP.NET MVC framework. This chapter also shows you how to share data between the ASP.NET MVC and ASP.NET Webforms.

Chapter 8 describes how you can use AJAX in combination with ASP.NET MVC by using two of the most popular AJAX frameworks: ASP.NET AJAX and jQuery. jQuery UI plugins are used to enrich ASP.NET MVC views.

Chapter 9 describes how you can create unit tests for your ASP.NET MVC applications, and explains what mocking is, and how this can help you when creating tests for an ASP.NET MVC application.

Chapter 10 describes how you can deploy and host an ASP.NET MVC application on the Internet Information Server (IIS6 and IIS7). You'll also see the differences between IIS' integrated mode and classic mode.

Appendix A builds a sample application, CarTrackr—an online software application designed to help you understand and track your fuel usage and kilometers driven. We will zoom in on certain aspects of this application, which will make your development of ASP.NET MVC applications easier and faster.

Appendix B contains source code that assists in testing an ASP.NET MVC application using a mocking framework, as described in Chapter 9 of this book. Source code is provided for use with three different mocking frameworks: RhinoMocks, Moq, and TypeMock.

Appendix C contains links to web sites that provide information and resources related to the ASP.NET MVC framework. It also examines several open source projects that provide additional features.

What you need for this book

No previous experience of the ASP.NET MVC framework is required. Because the ASP.NET MVC framework builds on top of ASP.NET, some previous experience with ASP.NET Webforms is useful in order to quickly catch up with the concepts that exist in ASP.NET Webforms and ASP.NET MVC. An understanding of JavaScript, HTML, and CSS is assumed, as well as an understanding of .NET 3.5 LINQ (Language Integrated Query).

Who this book is for

This book is for web developers with a basic knowledge of ASP.NET and C#, who wish to start using the new ASP.NET MVC framework.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "All the examples in this topic again make use of the `Contact` class defined earlier."

A block of code is set as follows:

```
public ActionResult UpdateContact(int id, string name, string email)
{
    Contact contact = Contacts.Single(c => c.Id == id);
    contact.Name = name;
    contact.Email = email;

    return RedirectToAction("Index");
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are shown in bold, as in the example below:

```
container.RegisterType<ICostsRepository, CostsRepository>(
    new ContextLifetimeManager<ICostsRepository>());

// Set controller factory
ControllerBuilder.Current.SetControllerFactory(
    new UnityControllerFactory(container))
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: " By default, views are located inside the **Views | ControllerName** project folder ".

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to feedback@packtpub.com, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or send an email to suggest@packtpub.com.

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book on, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit http://www.packtpub.com/files/code/7542_Code.zip to directly download the example code used in this book.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

ASP.NET MVC

Over the years, people have been asking the ASP.NET support team for the ability to develop web applications using a **model-view-controller (MVC)** architecture. In October 2007, Scott Guthrie presented the first preview of the ASP.NET MVC framework. Ever since, the interest in this product has been growing, and a vast amount of example applications, components, and so on have been released on the Internet by enthusiast bloggers and Microsoft employees.

This chapter describes the MVC pattern and explains the reason why Microsoft started the ASP.NET MVC framework project. It also compares ASP.NET MVC with ASP.NET Webforms and provides guidance on choosing between the two approaches for professional web development.

You will learn the following in this chapter:

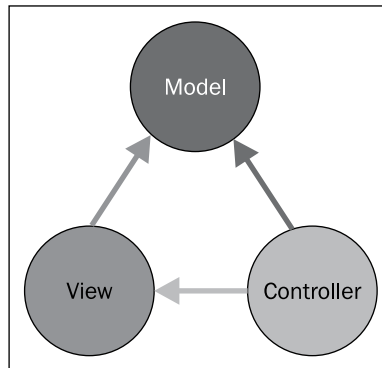
- What the **model-view-controller pattern** is, why it exists, and what its advantages are
- How the model-view-controller pattern is implemented in the ASP.NET MVC framework
- What the driving goals behind the ASP.NET MVC framework are
- How the ASP.NET MVC framework compares with ASP.NET Webforms
- How to choose between the two alternatives for ASP.NET web development

Model-view-controller

Model-view-controller, or MVC in short, is a design pattern used in software engineering. The main purpose of this design pattern is to isolate business logic from the user interface, in order to focus on better maintainability, testability, and a cleaner structure to the application. The MVC pattern consists of three key parts: the model, the controller, and the view.

Model

The model consists of some encapsulated data along with its processing logic, and is isolated from the manipulation logic, which is encapsulated in the controller. The presentation logic is located in the view component.



The model object knows all of the data that needs to be displayed. It can also define some operations that manipulate this encapsulated data. It knows absolutely nothing about the graphical user interface—it has no clue about how to display data or respond to actions that occur in the GUI. An example of a model would be a calculator. A calculator contains data (a numeric value), some methods to manipulate this data (add, subtract, multiply, and divide), and a method to retrieve the current value from this model.

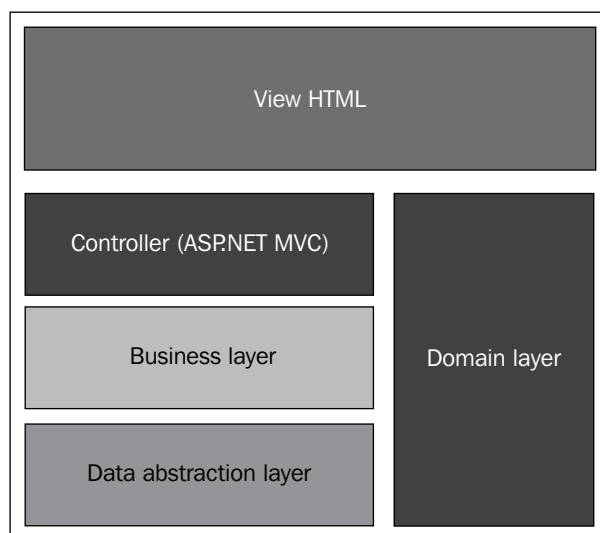
View

The view object refers to the model. It uses read-only methods of the model to query and retrieve data. It can look like an HTML page, a Windows GUI, or even the LED display of a physical calculator. In our example of the calculator, the view is the display of the calculator, which receives model data (the current calculation result) from the controller.

Controller

The controller object is the interaction glue of the model and the view. It knows that the model expects actions such as add, subtract, multiply, and divide, and also knows that the GUI will send some events that may require these operations to be called. In the calculator example, clicking on the "+" button on the view will trigger the controller to call the add method on the model and re-render the view with the data updated if necessary.

Applications are commonly split into three separate layers: presentation, business logic, and data access. These layers typically share a set of domain objects, which represent all of the entities that the application can work with. The MVC design pattern fits into the presentation layer, where it handles a user's interaction (controller) with a specific model through a view. Any application can be built using the MVC design pattern, be it a Winforms application, web, PDA, or other such things. The ASP.NET MVC framework, however, focuses on web applications, where the view is rendered as HTML—the controller sits on the web server and communicates with the business layer using the domain model. The business layer communicates with the data abstraction layer, also using the domain model as a shared set of entities that are passed around in the application logic. The schematic overview of an application layer structure based on the MVC pattern can be seen in the following figure:



The ASP.NET MVC framework

Being a web developer, you will definitely relate to some, if not all, of the following pains. The first web applications that developers created were executable programs running on a server, which were called **Common Gateway Interface (CGI)** applications. These CGI programs accepted an HTTP request issued by a user's web browser, and returned HTML responses based on the requested action. One of the difficulties with these kinds of programs is that they are very hard to maintain and scale.

Along with other software companies, Microsoft started creating their own implementation for delivering interactive web applications, **Active Server Pages**, or **ASP**, at that time. One disadvantage of ASP was that code and markup were sitting in the same file, making the code very hard to read and maintain, especially for larger projects. Luckily, ASP.NET was introduced a few years later. ASP.NET offered the separation of code and templates, and allowed for easier debugging and rapid application development by using an event-driven model that most desktop developers are familiar with. For example, ASP.NET provides an `OnClick` event handler, which is fired after a user clicks on a button, in the same way that Winforms development is done.

At the end of 2007, Microsoft released a first preview of the ASP.NET MVC framework that would break with dependencies on this event-driven model and allow for cleaner separation of model, code, and markup.

Driving goals of the ASP.NET MVC framework

Microsoft started building the ASP.NET MVC framework with the following ideas in mind:

- Clean separation of concerns, testability, and support for **test-driven development (TDD)** by default. The framework provides interface-based and thus easily mockable core contracts. Unit tests are not required to be run inside an ASP.NET process, making unit testing fast and independent of a specific unit test framework (NUnit, MS Test, xUnit, and so on). In ASP.NET Webforms, testing could be done only after deploying an application and database on a server and firing automated macros on the user interface. In ASP.NET MVC, every action that a user can perform can be unit tested automatically, without requiring the deployment of the application.
- A highly extensible and pluggable architecture – every component can be easily replaced or customized. This pluggable architecture also allows easier use of the dependency injection design pattern by using frameworks such as Unity, Castle Windsor, Spring.net, and so on.
- It includes a powerful URL routing component that enables you to build applications with clean, search engine friendly URLs. For example, the URL `/employees/show/123` could be easily mapped to the `Show` action method of the `EmployeesController` class for employee number 123.
- Existing ASP.NET features are still available: master pages, content pages, user controls, sessions, membership, and so on. The only difference is that there's no ViewState or page life cycle involved.

- Full control of your HTML markup. The ASP.NET MVC framework does not inject extraneous HTML code into your rendered page, unlike ASP.NET Webforms, which injects ViewState, resources, and so on.

The MVC pattern helps you to create applications that have a clean separation of concerns. Separation of concerns specifies where each type of logic should be located in the application. This helps you to manage complexity and scalability when building an application. As the application is divided into different modules (data, interaction, user interface, and so on), it becomes easier to maintain and test. The separation of all of the components also allows for parallel development: one developer can work on the model, another one on the controller, and a web designer can focus on creating the view. Using the ASP.NET framework enables you to make extensive use of the advantages that the MVC pattern offers.



Aside from using the model-view-controller pattern provided in the ASP.NET MVC framework, the **Microsoft Patterns & Practices Team** provides the **Web Client Software Factory (WCSF)** to help implement the **model-view-presenter (MVP)** design pattern in your applications.

The MVC and MVP patterns are comparable, but differ in certain aspects. The view in MVC knows about the model, whereas the view in MVP does not. In MVC, the view is responsible for how model data is represented, whereas in the MVP pattern, the presenter sets data in the view.

Another difference between both patterns is that in the MVC pattern, there are multiple controllers, whereas presenters in the MVP pattern are usually responsible for all of the page flows regarding a certain subject. For example, the MVC pattern might have a `PricingController` and a `CustomerController`, whereas in the MVP pattern, these can be grouped in a `SalesPresenter`.

Refer to the following URL for more information on the differences between ASP.NET MVC and WCSF: <http://blogs.msdn.com/simonince/archive/2007/11/22/the-asp-net-mvc-framework-using-the-wcsf-as-a-yardstick.aspx>

Comparing ASP.NET MVC and ASP.NET Webforms

You should know that the ASP.NET MVC framework is not a replacement for ASP.NET Webforms—it's an alternative that you can choose if it is well-suited for a specific situation. Make sure that you weigh and compare the advantages of each solution prior to picking a specific direction.

The ASP.NET MVC framework offers the following advantages:

- Complexity of application logic is made easier to manage because of the separation of an application into model, view, and controller.
- It allows for easier parallel development; each developer can work on a separate component of the MVC pattern.
- It provides good support for TDD, mocking, and unit testing frameworks. TDD enables you to write tests for an application first, after which the application logic is developed. TDD, mocking, and unit testing are explained in Chapter 9, *Testing an Application*.
- It does not use ViewState or server-based forms, which allows you to have full control over the application's behavior and HTML markup.
- It uses RESTful interfaces for URLs, which is better for **SEO (Search Engine Optimization)**. **REST** is short for **REpresentational State Transfer** – the concept of using URLs as the link to a resource, which can be a controller action method, rather than to physical files on the web server.
- A typical page size is small, because no unnecessary data is transferred in the form of hidden ViewState data.
- It integrates easily with client-side JavaScript frameworks such as jQuery or ExtJS.

ASP.NET Webforms offers the following advantages:

- It offers an event model over HTTP that is familiar to any developer. This event model also benefits the creation of business web applications.
- It provides a lot of controls that are familiar to any developer – data components such as data grids and lists, validation controls, and so on. These components are highly integrated in the development environment.
- There are numerous third-party control vendors that can deliver almost any possible control.
- Being familiar to developers allows ASP.NET Webforms to facilitate rapid application development.
- Functionality is concentrated per page. It uses ViewState and server-based forms, which makes state management easier.

Choosing between ASP.NET MVC and ASP.NET Webforms

In general, choosing between ASP.NET MVC and ASP.NET can be based on the following five criteria:

1. **Are you considering test-driven development (TDD)?**
TDD enables you to write tests for an application first, after which the application logic is developed. An ASP.NET Webforms application uses only one class to display output and handle user input. Automated testing of this class is complex as you are required to load the full ASP.NET stack into a separate process (for example, in IIS). The MVC framework allows the testing of each component separately, without requiring the full ASP.NET stack. For example, you can test your model separately from your controller without requiring a view. If you are considering TDD, the ASP.NET MVC framework will be the better choice.
2. **Is there a need for fine control over HTML markup?**
ASP.NET Webforms automatically inserts hidden HTML markup, IDs, JavaScript, and so on, into your page's HTML output because of its event-driven architecture and its use of ViewState. The ASP.NET MVC framework allows for 100% control over the HTML output. If you require full control over your HTML markup, the ASP.NET MVC framework will be the better choice.
3. **Is the application heavily data-driven?**
If you are developing an application that is heavily data-driven and is using grids or a lot of master-detail editing of data, ASP.NET Webforms may be the better choice as it provides a lot of controls that will aid you in the development of these kind of applications. Of course, you can use the ASP.NET MVC framework for these tasks too, but you will be required to write more code to achieve the same goal.
4. **Is there a need for a Winforms-like development approach?**
Does your development team write Winforms code? Is there a need for an event-driven programming approach? In these cases, consider ASP.NET Webforms in favor of ASP.NET MVC.
5. **Is there a need for a rapid application development (RAD) development approach?**
Does your client expect a quick prototype of an application? Is the use of drag-and-drop development using pre-created web controls required? If so, consider ASP.NET Webforms in favor of ASP.NET MVC.

Summary

In this chapter, we have learned what the model-view-controller pattern is, why it is there, and what its advantages are. We also have seen how this pattern is the base for the ASP.NET MVC framework and what the driving goals behind the ASP.NET MVC framework are.

Another thing that we have seen is how the ASP.NET MVC framework compares with ASP.NET Webforms, and also how to choose between the two alternatives in ASP.NET web development.

2

Your First ASP.NET MVC Application

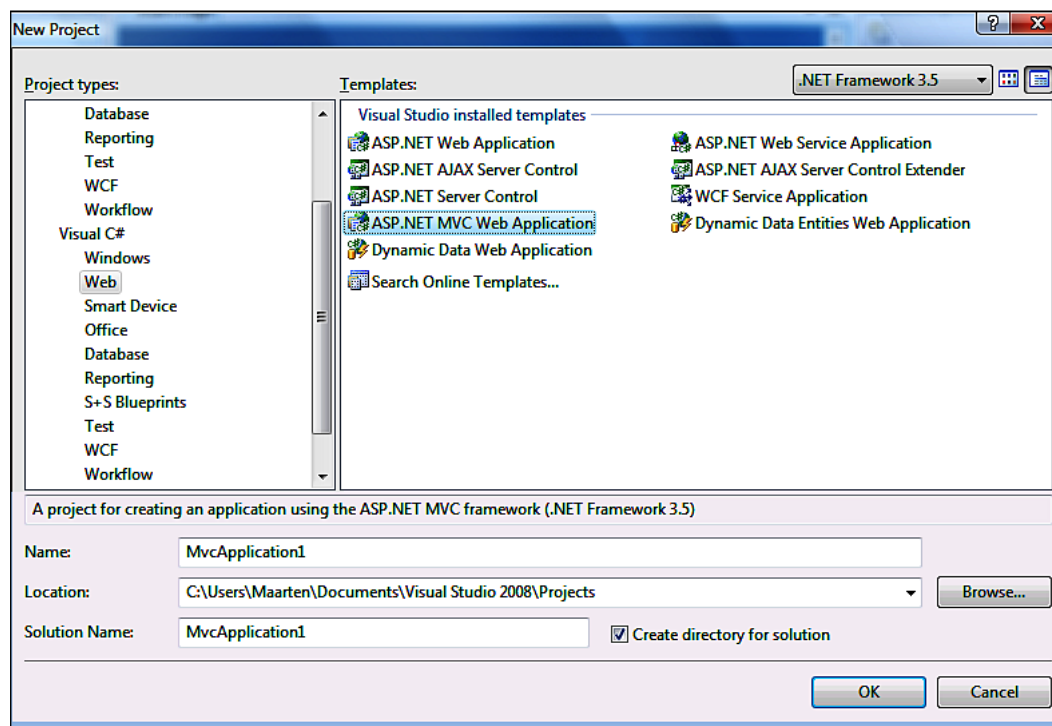
When downloading and installing the ASP.NET MVC framework SDK, a new project template is installed in Visual Studio – the ASP.NET MVC project template. This chapter describes how to use this template. We will briefly touch all aspects of ASP.NET MVC by creating a new ASP.NET MVC web application based on this Visual Studio template. Besides view, controller, and model, new concepts including ViewData – a means of transferring data between controller and view, routing – the link between a web browser URL and a specific action method inside a controller, and unit testing of a controller are also illustrated in this chapter.

In this chapter, you will:

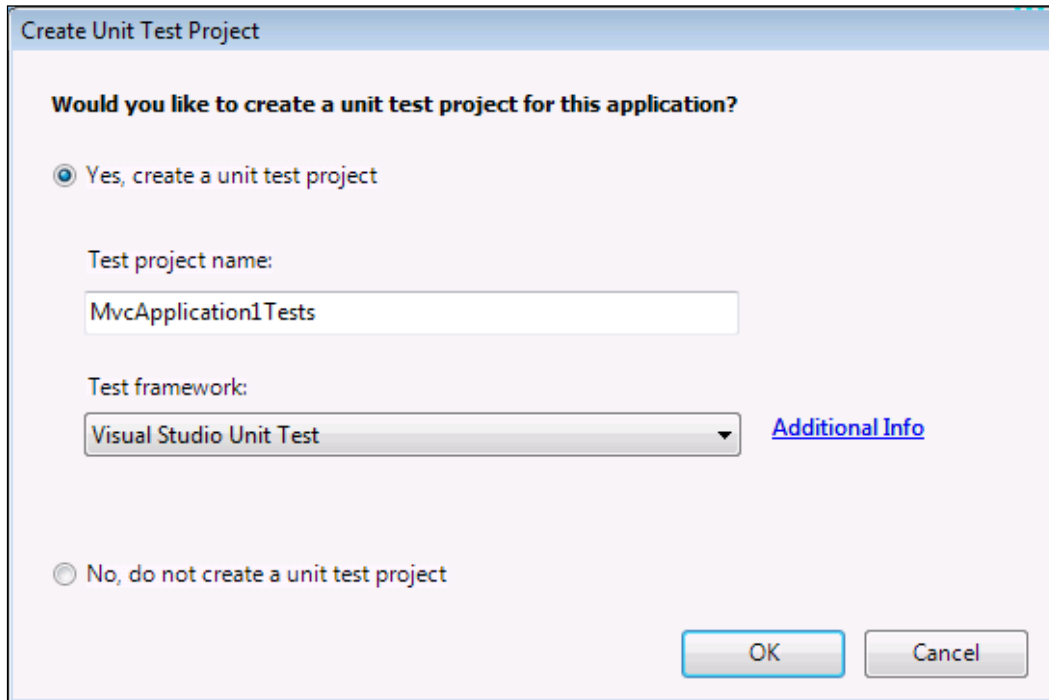
- Receive an overview of all of the aspects of an ASP.NET MVC web application
- Explore the ASP.NET MVC web application project template that is installed in Visual Studio 2008
- Create a first action method and a corresponding view
- Create a strong-typed view
- Learn how a controller action method can pass strong-typed ViewData to the view
- Learn what unit testing is all about, and why it should be performed
- Learn how to create a unit test for an action method by using Visual Studio's unit test generation wizard and modifying the unit test code by hand

Creating a new ASP.NET MVC web application project

Before we start creating an ASP.NET MVC web application, make sure that you have installed the ASP.NET MVC framework SDK from www.asp.net/mvc. After installation, open Visual Studio 2008 and select menu option **File | New | Project**. The following screenshot will be displayed. Make sure that you select the **.NET framework 3.5** as the target framework. You will notice a new project template called **ASP.NET MVC Web Application**. This project template creates the default project structure for an ASP.NET MVC application.



After clicking on **OK**, Visual Studio will ask you if you want to create a test project. This dialog offers the choice between several unit testing frameworks that can be used for testing your ASP.NET MVC application.

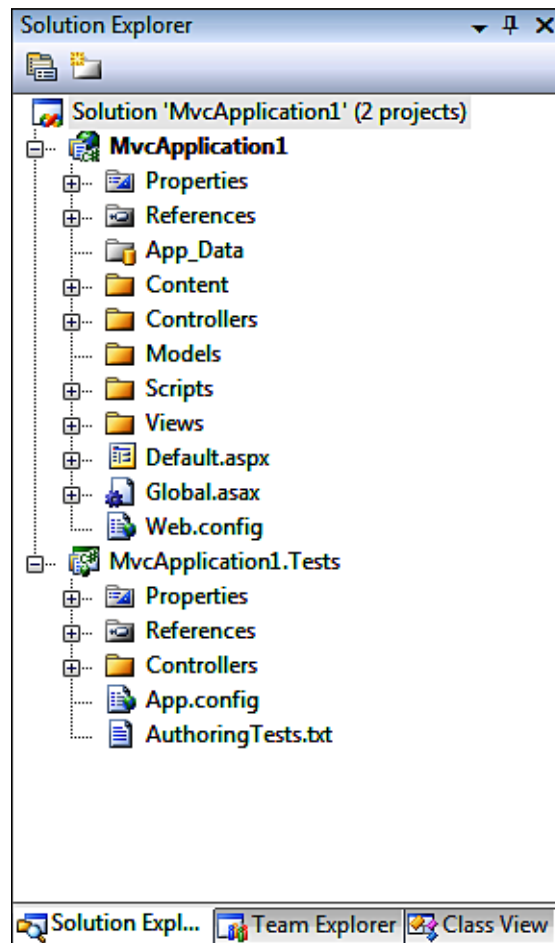


The image shows a 'Create Unit Test Project' dialog box. At the top, it asks 'Would you like to create a unit test project for this application?'. There are two radio buttons: 'Yes, create a unit test project' (which is selected) and 'No, do not create a unit test project'. Below the 'Yes' option, there is a text box for 'Test project name:' containing 'MvcApplication1Tests'. Below that is a dropdown menu for 'Test framework:' with 'Visual Studio Unit Test' selected. To the right of the dropdown is a blue link labeled 'Additional Info'. At the bottom right are 'OK' and 'Cancel' buttons.

You can decide for yourself if you want to create a unit testing project right now – you can also add a testing project later on. Letting the ASP.NET MVC project template create a test project now is convenient because it creates all of the project references, and contains an example unit test, although this is not required. For this example, continue by adding the default unit test project.

What's inside the box?

After the ASP.NET MVC project has been created, you will notice a default folder structure. There's a **Controllers** folder, a **Models** folder, a **Views** folder, as well as a **Content** folder and a **Scripts** folder. ASP.NET MVC comes with the convention that these folders (and namespaces) are used for locating the different blocks used for building the ASP.NET MVC framework. The **Controllers** folder obviously contains all of the controller classes; the **Models** folder contains the model classes; while the **Views** folder contains the view pages. **Content** will typically contain web site content such as images and stylesheet files, and **Scripts** will contain all of the JavaScript files used by the web application. By default, the **Scripts** folder contains some JavaScript files required for the use of Microsoft AJAX or jQuery.



Locating the different building blocks is done in the request life cycle, which is described in Chapter 4, *Components in the ASP.NET MVC Framework*. One of the first steps in the ASP.NET MVC request life cycle is mapping the requested URL to the correct controller action method. This process is referred to as **routing**. A default route is initialized in the `Global.asax` file and describes to the ASP.NET MVC framework how to handle a request. Double-clicking on the `Global.asax` file in the `MvcApplication1` project will display the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class GlobalApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index",
                    id = "" } // Parameter defaults
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

In the `Application_Start()` event handler, which is fired whenever the application is compiled or the web server is restarted, a route table is registered. The default route is named `Default`, and responds to a URL in the form of `http://www.example.com/{controller}/{action}/{id}`. The variables between `{` and `}` are populated with actual values from the request URL or with the default values if no override is present in the URL. This default route will map to the `Home` controller and to the `Index` action method, according to the default routing parameters. We won't have any other action with this routing map.

By default, all possible URLs can be mapped through this default route. It is also possible to create our own routes. For example, let's map the URL `http://www.example.com/Employee/Maarten` to the `Employee` controller, the `Show` action, and the `firstname` parameter. The following code snippet can be inserted in the `Global.asax` file we've just opened. Because the ASP.NET MVC framework uses the first matching route, this code snippet should be inserted above the default route; otherwise the route will never be used.

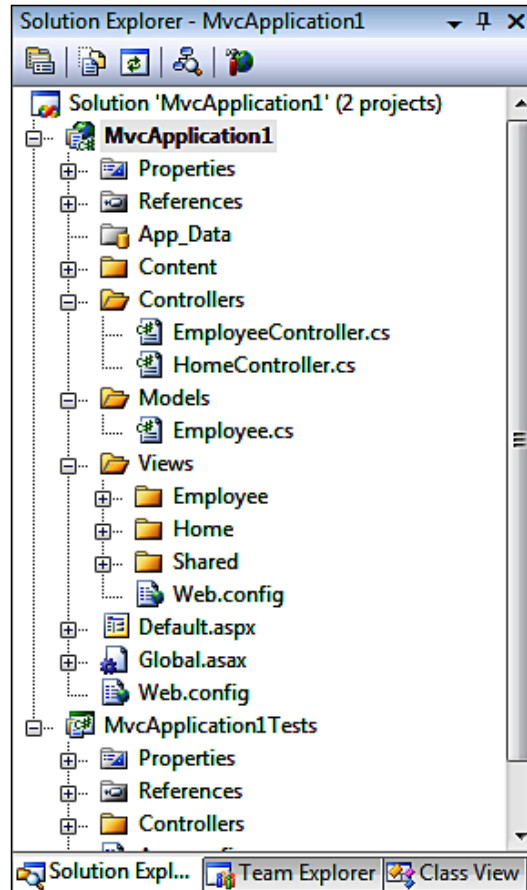
```
routes.MapRoute(
    "EmployeeShow",                // Route name
    "Employee/{firstname}",        // URL with parameters
    new {                          // Parameter defaults
        controller = "Employee",
        action = "Show",
        firstname = ""
    }
);
```

Now, let's add the necessary components for this route. First of all, create a class named `EmployeeController` in the `/Controllers` folder. You can do this by adding a new item to the project and selecting the **MVC Controller Class** template located under the **Web | MVC** category. Remove the `Index` action method, and replace it with a method or action named `Show`. This method accepts a `firstname` parameter and passes the data into the `ViewData` dictionary. This dictionary will be used by the view to display data.

The `EmployeeController` class will pass an `Employee` object to the view. This `Employee` class should be added in the **Models** folder (right-click on this folder and then select **Add | Class** from the context menu). Here's the code for the `Employee` class:

```
namespace MvcApplication1.Models
{
    public class Employee
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
    }
}
```

After adding the `EmployeeController` and `Employee` classes, the ASP.NET MVC project now appears as shown in the following screenshot:



The `EmployeeController` class now looks like this:

```
using System.Web.Mvc;
using MvcApplication1.Models;
namespace MvcApplication1.Controllers
{
    public class EmployeeController : Controller
    {
        public ActionResult Show(string firstname)
        {
            if (string.IsNullOrEmpty(firstname))
            {
```

```
        ViewData["ErrorMessage"] = "No firstname provided!";
    }
    else
    {
        Employee employee = new Employee
        {
            FirstName = firstname,
            LastName = "Example",
            Email = firstname + "@example.com"
        };
        ViewData["FirstName"] = employee.FirstName;
        ViewData["LastName"] = employee.LastName;
        ViewData["Email"] = employee.Email;
    }
    return View();
}
}
```

The action method we've just created can be requested by a user via a URL—in this case, something similar to `http://www.example.com/Employee/Maarten`. This URL is mapped to the action method by the route we've created before.

By default, any public action method (that is, a method in a controller class) can be requested using the default routing scheme. If you want to avoid a method from being requested, simply make it private or protected, or if it has to be public, add a `[NonAction]` attribute to the method.

Note that we are returning an `ActionResult` (created by the `View()` method), which can be a view-rendering command, a page redirect, a JSON result (discussed in detail in Chapter 8, *AJAX and ASP.NET MVC*), a string, or any other custom class implementation inheriting the `ActionResult` that you want to return. Returning an `ActionResult` is not necessary. The controller can write content directly to the response stream if required, but this would be breaking the MVC pattern—the controller should never be responsible for the actual content of the response that is being returned.

Next, create a `Show.aspx` page in the **Views | Employee** folder. You can create a view by adding a new item to the project and selecting the **MVC View Content Page** template, located under the **Web | MVC** category, as we want this view to render in a master page (located in **Views | Shared**). There is an alternative way to create a view related to an action method, which will be covered later in this chapter.

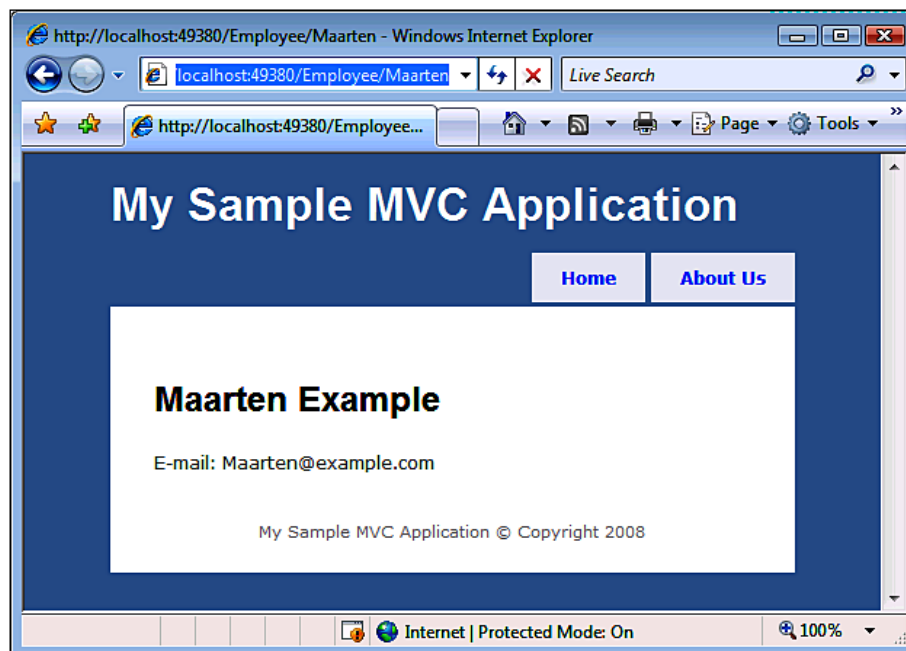
In the view, you can display employee information or display an error message if an employee is not found.

Add the following code to the Show.aspx page:

```
<%@ Page Title="" Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
    AutoEventWireup="true" Inherits="System.Web.Mvc.ViewPage" %>
<asp:Content ID="Content1"
    ContentPlaceHolderID="MainContent"
    runat="server">
    <% if (ViewData["ErrorMessage"] != null) { %>
    <h1><%=ViewData["ErrorMessage"]%></h1>
    <% } else { %>
    <h1><%=ViewData["FirstName"]%> <%=ViewData["LastName"]%></h1>
    <p>
        E-mail: <%=ViewData["Email"]%>
    </p>
    <% } %>
</asp:Content>
```

If the ViewData, set by the controller, is given an ErrorMessage, then the ErrorMessage is displayed on the resulting web page. Otherwise, the employee details are displayed.

Press the F5 button on your keyboard to start the development web server. Alter the URL in your browser to something ending in /Employee/Your_Name_Here, and see the action method and the view we've just created in action.



Strong-typed ViewData

In the previous example, we used the `ViewData` dictionary to pass data from the controller to the view. When developing the view, each dictionary item we want to display should be cast to the correct class, resulting in a less maintainable situation. It might also lead to code spaghetti in the view. It would be useful if the `ViewData` dictionary already knew which class type each of its items represented. This is where the **model** comes in handy! We are serving employee information to the view, so why not use the `Employee` class that we'd previously created as a "the" model for our view? Note that we'd already placed the `Employee` class inside the **Model** folder, which is the appropriate location for model classes.

Views can be made strong-typed by updating the view and replacing the base class of the page (`System.Web.Mvc.ViewPage`) with a generic version: `System.Web.Mvc.ViewPage<Employee>`. Make sure you compile your project after updating the first few lines of code in the `Show.aspx` file:

```
<%@ Page Title=""
    Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
    AutoEventWireup="true"
    Inherits="System.Web.Mvc.ViewPage<MvcApplication1.Models.
        Employee>" %>
```

By applying the above code, the page's `ViewData` object will be made generic. This means that the `ViewData` object will not only be a dictionary, but will also contain a property named `Model`, which is of the type that has just been passed in: `MvcApplication1.Models.Employee`.

This `ViewData.Model` property is also available as a `Model` property in the view. We will have to update the view to be able to use this new property. Simply change from `ViewData[key]` to a property `Model` (which contains the `Employee` instance). For example, `Model.FirstName` is in fact the `FirstName` property of the `Employee` instance that you want to render. Note that you can still use dictionary entries combined with this strong-typed model.

```
<%@ Page Title=""
    Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
    AutoEventWireup="true"
    Inherits="System.Web.Mvc.ViewPage<MvcApplication1.Models.
        Employee>" %>
<asp:Content ID="Content1"
    ContentPlaceHolderID="MainContent"
    runat="server">
```

```

    <% if (ViewData["ErrorMessage"] != null) { %>
    <h1><%=ViewData["ErrorMessage"]%></h1>
    <% } else { %>
    <h1><%=Model.FirstName%> <%=ViewData.Model.LastName%></h1>
    <p>
        E-mail: <%=Model.Email%>
    </p>
    <% } %>
</asp:Content>

```

Before being able to run the application, the controller needs some updates as well. The main difference is that employee properties are no longer copied into the ViewData dictionary. Instead, the `Employee` instance is passed directly to the view.

```

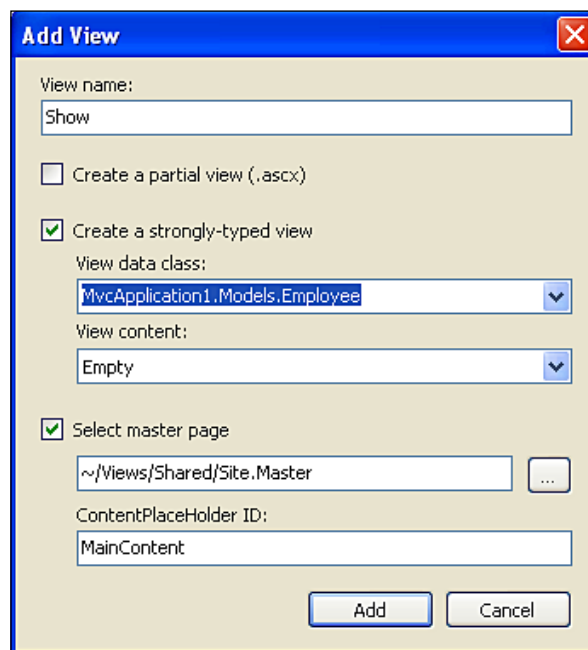
using System.Web.Mvc;
using MvcApplication1.Models;
namespace MvcApplication1.Controllers
{
    public class EmployeeController : Controller
    {
        public ActionResult Show(string firstname)
        {
            Employee employee = null;
            if (string.IsNullOrEmpty(firstname))
            {
                ViewData["ErrorMessage"] = "No firstname provided!";
            }
            else
            {
                employee = new Employee
                {
                    FirstName = firstname,
                    LastName = "Example",
                    Email = firstname + "@example.com"
                };
            }
            return View(employee);
        }
    }
}

```

Note that we are passing the model data to the `View()` method. Alternatively, this can be done by stating `Model = employee` prior to returning the view. If you run the application, the result should be exactly the same as before.

Creating a new view

During the development of the controller action method, creating a corresponding view is very straightforward. To create a new view for the current controller action, right-click somewhere on the method body, and select **Add view...** from the context menu. The following dialog box will be displayed:

The image shows the 'Add View' dialog box in Visual Studio. It has a blue title bar with the text 'Add View' and a close button. The dialog contains several fields and checkboxes. The 'View name:' field has 'Show' entered. There are two checkboxes: 'Create a partial view (.ascx)' which is unchecked, and 'Create a strongly-typed view' which is checked. Below the second checkbox is a 'View data class:' dropdown menu showing 'MvcApplication1.Models.Employee'. The 'View content:' dropdown menu shows 'Empty'. There is a checked checkbox for 'Select master page' with a text field below it containing '~\Views\Shared\Site.Master' and a browse button (...). At the bottom is a 'ContentPlaceHolder ID:' field with 'MainContent' entered. At the very bottom are 'Add' and 'Cancel' buttons.

In the **Add view** dialog box, some options can be specified. First of all, the view name can be modified if required. By default, this name will be the same as the action method name. It's also possible to select a view template, which we will set to **Empty**. This template can be used to easily create a view — for example, one which shows the details of an employee. You will see a little more about this in Chapter 4, *Components in the ASP.NET MVC Framework*.

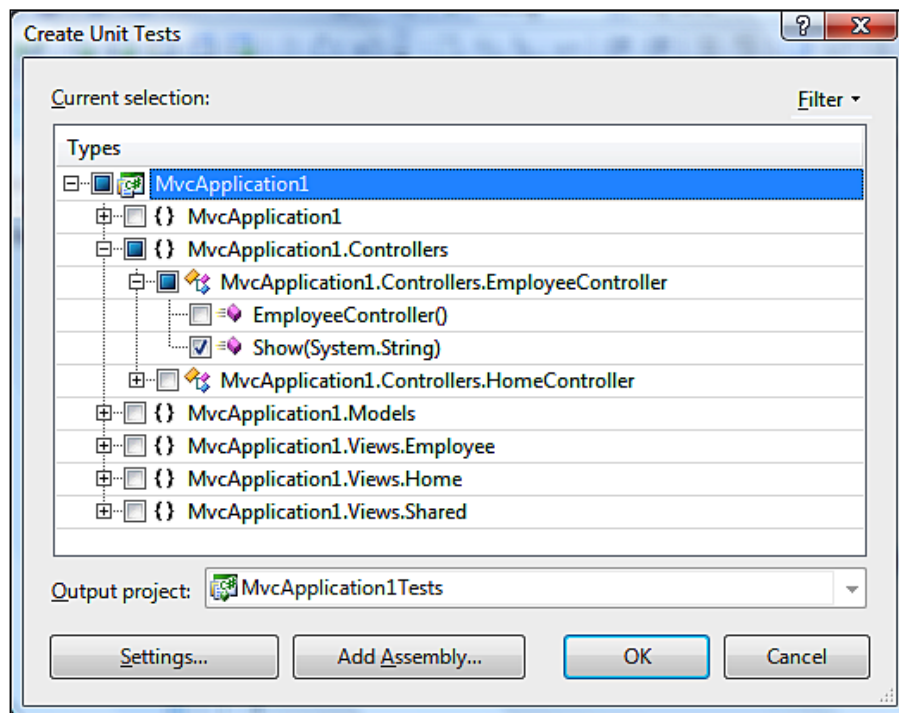
From this dialog, it's also possible to make the view strongly-typed by simply selecting the corresponding checkbox and choosing the class to base the view on. The last option in this dialog box allows you to specify the master page.

Unit testing the controller

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently tested for correct operation. Typically, these **units** are individual methods being tested. Most often, unit tests are run automatically, and provide immediate feedback (successful/unsuccessful/unknown result) to a developer on the changes he or she has just made to the code. If a test is unsuccessful, the changes to the code should be reviewed because the expected behavior of a portion of source code has changed and may affect other units or the application as a whole.

When we created the ASP.NET MVC web application, a test project was also created. This already contains an example test class for `HomeController`, testing both the `Index` and `About` actions.

In the **MvcApplication1Tests** project, right-click on the **Controllers** folder, and then select **Add | Unit Test** from the context menu. From the wizard that is displayed, select the **Show** method of `EmployeeController` and click on **OK**. Visual Studio will generate a test class.



Modify the generated test class to look like the following code:

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MvcApplication1.Controllers;
namespace MvcApplication1Tests.Controllers
{
    /// <summary>
    /// Summary description for EmployeeControllerTest
    /// </summary>
    [TestClass]
    public class EmployeeControllerTest
    {
        [TestMethod]
        public void show_action_creates_employee_and_passes_to
            _view_when_firstname_is_specified()
        {
            // Setup
            EmployeeController controller = new EmployeeController();
            // Execute
            ViewResult result = controller.Show("Maarten") as
                ViewResult;

            // Verify
            Assert.IsNotNull(result);
            ViewDataDictionary viewData = result.ViewData;
            Assert.IsNotNull(viewData.Model);
            Assert.AreEqual("Maarten", (viewData.Model as
                MvcApplication1.Models.Employee).FirstName);
            Assert.IsNull(viewData["ErrorMessage"]);
        }

        [TestMethod]
        public void show_action_passes_error_model_to
            _view_when_no_firstname_specified()
        {
            // Setup
            EmployeeController controller = new EmployeeController();
            // Execute
            ViewResult result = controller.Show(null) as ViewResult;
            // Verify
            Assert.IsNotNull(result);
            ViewDataDictionary viewData = result.ViewData;
            Assert.IsNull(viewData.Model);
            Assert.IsNotNull(viewData["ErrorMessage"]);
        }
    }
}
```

Each test method is first initializing a new `EmployeeController`, after which the action method that needs to be tested is called. When the action method returns an `ActionResult`, it is cast to `ViewResult` on which some assertions are made. For example, if the `Show` action method of `EmployeeController` is called with parameter `Maarten`, an assertion is made that the controller passes the correct employee data to the view.

This test suite does not require the application to be deployed on a web server. Instead, the `EmployeeController` is tested directly in the code. The test asserts that some properties of the `ViewData` are present. For example, if the `Show()` action is called with the parameter, `Maarten`, the `Model` should not be null and should contain an `Employee` with the first name, `Maarten`.



More advanced testing scenarios are explained in Chapter 9, *Testing an Application*.

Summary

In this chapter, we have seen an overview of all aspects of an ASP.NET MVC web application. We started by exploring the ASP.NET MVC web application project template that is installed in Visual Studio 2008, after which we created our own action method and corresponding view.

Another thing that we have seen is how to create a strong-typed view and how a controller action method can pass strong-typed `ViewData` to the view.

We ended this chapter by looking at the various aspects of unit testing—what it is, why it is used, how to create a unit test for an action method by using Visual Studio's unit test generation wizard, and modifying the unit test code by hand.

3

Handling Interactions

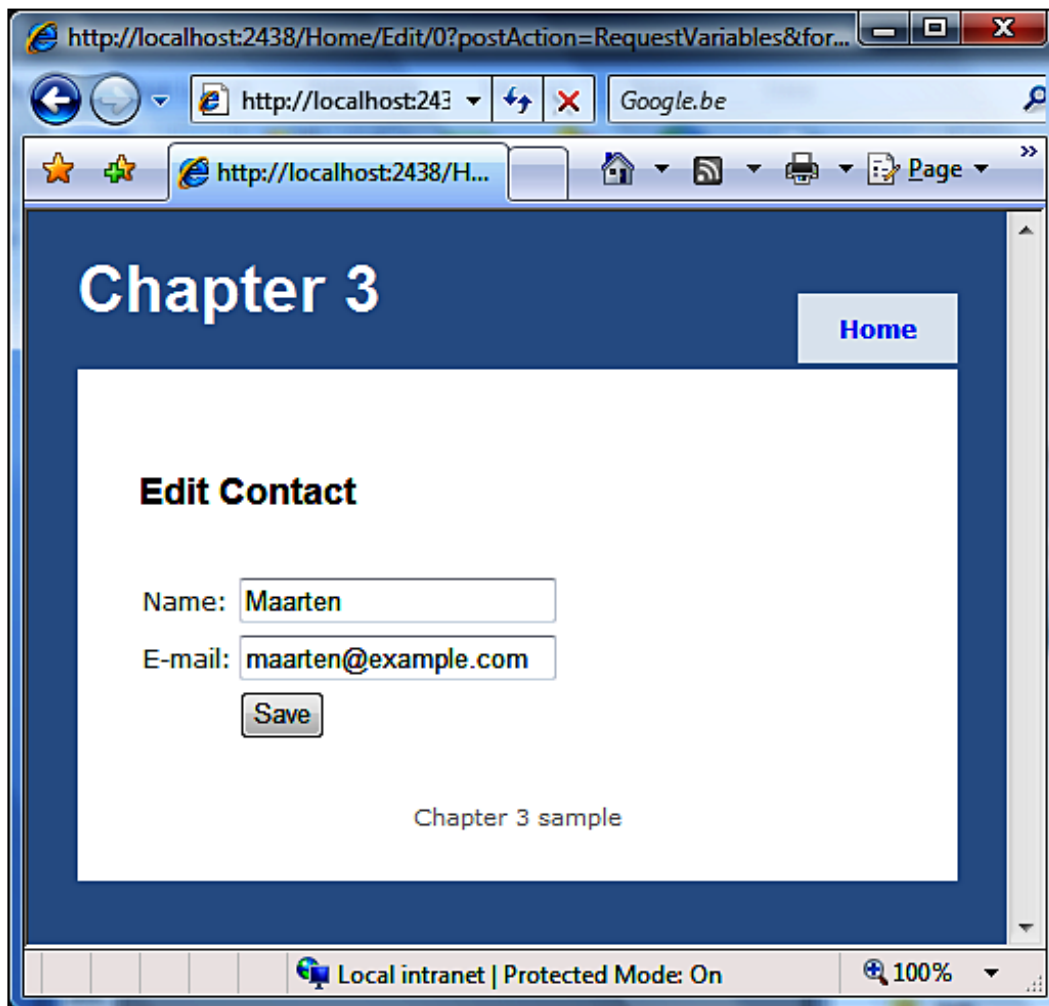
When using a modern web application, there may be times when a user may have to fill out a form and post it to your server. ASP.NET Webforms offer a layer of abstraction around HTML forms that maintains a ViewState and provides an easy interface to form elements. With the ASP.NET MVC framework, this is slightly different as it has to be done in pure HTML, although an `HtmlHelper` class is available to assist you. This chapter will guide you through the process of creating a frontend form and responding to posts in the controller.

You will learn the following in this chapter:

- The different methods that exist to create an HTML form
- Posting data to an action method by making use of HTML and the `HtmlHelper` class
- Reading values from a form post in an action method by using request variables and action method parameters
- Handling file uploads in an action method
- An overview of the model binder infrastructure provided by the ASP.NET MVC framework
- Implementing a custom `IMoDelBinder`
- Validation of data
- Providing feedback to the user of our web application.

This chapter builds a small application that lets you edit contact details (name and email address). These contact details will be provided in a class named `Contact`, which will be used throughout this chapter:

```
public class Contact
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```



Creating a form

Creating a form in which the user of your web application can enter some data is something that you are likely to do. Nowadays, almost every web application you see contains several forms that collect user data that is processed by the web server.

This topic explains two different approaches to creating a form. The first approach requires you to build a form by using pure HTML markup. The second approach uses the `HtmlHelper` class that is provided by the ASP.NET MVC framework.

`HtmlHelper` offers a standard set of helper methods that provide a programmatic method for creating HTML controls.

The examples in this topic are based on an ASP.NET MVC web application that can be found in the sample code for this book (UpdatingDataExample). This sample project contains one controller on which two different views depend. In this topic, only the view portion of this example will be covered, as the controller part will be covered later in this chapter.

Creating a form using HTML

The most basic method of creating a view containing a form is creating it in pure HTML. One can simply add a `<form>` tag and include the necessary form fields that can be updated by a user running the application.

```
<h2>Edit Contact</h2>
<form method="post" action="/Home/ UpdatingObjects">
  <input type="hidden" name="Id"
    value="<%=Html.Encode(Model..Id.ToString())%>" />
  <table border="0" cellspacing="0" cellpadding="2">
    <tr>
      <td>Name:</td>
      <td><input type="text" name="Name"
        value="<%=Html.Encode (Model..Name) %>" /></td>
    </tr>
    <tr>
      <td>E-mail:</td>
      <td><input type="text" name="Email"
        value="<%=Html.Encode (Model..Email) %>" /></td>
    </tr>
    <tr>
      <td>&nbsp;</td>
      <td><input type="submit"
        value="Save">
    </td>
  </table>
</form>
```

```
                                name="saveButton" />
                                </td>
                            </tr>
                        </table>
                    </form>
```

This code will render as a form containing a table in which properties such as a name and email address can be edited.

One thing to notice is that HTML and ASP control tags are mixed and may result in code spaghetti when building large forms. This can be addressed by using the `HtmlHelper` methods. We will see more of the `HtmlHelper` class in the next topic.



You may have noticed that this code sample is referring to `Model . Contact . Name` and `Model . Contact . Email` in the view. Why would `Contact` be a property of the model object? Isn't `Contact` the model object itself?

I always tend to make my model a custom class model containing data that is being passed to the view. In this case, I have an `EditContactViewData` class containing a `Contact` property that refers to the contact being edited. This approach may seem a little strange, but when more `ViewData` of different kinds needs to be passed to the view, I can simply add a new property to the `EditContactViewData` class, so that it provides a `Contact` property as well (for example) as an `Invoice` property. If we use `Contact` as the model, the chances are that we will have to do some refactoring when we have to pass more data to the view.

Creating a form using `HtmlHelper`

When building large, pure HTML forms, readability and maintainability of these forms can be tedious and hard. The ASP.NET MVC framework features a class called `HtmlHelper`, which provides each view with a standard set of helper methods that provide a programmatic method of creating HTML controls. When executing the application, `HtmlHelper` methods are rendered into plain HTML. Every view page provides an `Html` property, which is an instance of the `HtmlHelper` class.

```
<h2>Edit Contact</h2>

<% using (Html.BeginForm("UpdatingObjects ", "Home",
    FormMethod.Post)) { %>
    <%=Html.Hidden("Id"
    <table border="0" cellspacing="0" cellpadding="2">
        <tr>
            <td>Name:</td>
            <td><%=Html.TextBox("Name") %></td>
```

```

        </tr>
        <tr>
            <td>E-mail:</td>
            <td><%=Html.TextBox("Email")%></td>
        </tr>
        <tr>
            <td>&nbsp;</td>
            <td><input type="submit" value="Send e-mail" /></td>
        </tr>
    </table>
<% } %>

```

The following helper and extension methods are defined on the `HtmlHelper` class:

Method	Description
<code>ActionLink</code>	Generates a link to a specific controller action
<code>AntiForgeryToken</code>	Generates a hidden form field that can be used in conjunction with the <code>ValidateAntiForgeryToken</code> attribute to ensure that a request has not been modified
<code>AttributeEncode</code>	Encodes attribute data
<code>DropDownList</code>	Generates a drop-down list based on a list of key-value pairs
<code>Encode</code>	Encodes a string to prevent XSS
<code>EvalBoolean</code>	Searches for a specified key in the <code>ViewData</code> dictionary and returns the value as a <code>Boolean</code>
<code>EvalString</code>	Searches for a specified key in the <code>ViewData</code> dictionary and returns the value as a <code>String</code>
<code>Hidden</code>	Generates a hidden field
<code>ListBox</code>	Generates a list box based on a list of key-value pairs
<code>Password</code>	Generates a password field
<code>RouteLink</code>	Generates a link to a specific route
<code>TextBox</code>	Generates a text box
<code>Button</code>	Generates a button
<code>CheckBox</code>	Generates a checkbox
<code>BeginForm</code>	Generates an HTML form
<code>Image</code>	Generates an image tag
<code>Mailto</code>	Generates a mail-to hyperlink
<code>NavigateButton</code>	Generates a button that navigates to a specific URL
<code>RadioButton</code>	Generates an option button
<code>RenderAction</code>	Renders a controller action method

Method	Description
RenderPartial	Renders a partial view that can optionally be rendered by another view engine
RenderUserControl	Renders a user control
RouteLink	Generates a link to a specific route
SubmitButton	Renders a submit button
SubmitImage	Renders a submit image
TextArea	Renders a text area
ValidationMessage	Renders a validation message for a specific ViewData.ModelState key
ValidationSummary	Renders a validation summary for ViewData.ModelState

When working with strong-typed views, it is also possible to use the strong-typed `HtmlHelper` instance. For example, creating a text box that can be used for editing an Email property can be achieved using something like `Html.TextBoxFor(x => x.Email)`.

Here's the example we created previously using strong-typed `HtmlHelper`:

```
<h2>Edit Contact</h2>
<% using (Html.BeginForm("UpdatingObjects ", "Home",
    FormMethod.Post)) { %>
    <%=Html.Hidden("Id"
    <table border="0" cellspacing="0" cellpadding="2">
        <tr>
            <td>Name:</td>
            <td><%=Html.TextBoxFor(x => x.Name)%></td>
        </tr>
        <tr>
            <td>E-mail:</td>
            <td><%=Html.TextBox(x => x.Email)%></td>
        </tr>
        <tr>
            <td>&nbsp;</td>
            <td><input type="submit" value="Send e-mail" /></td>
        </tr>
    </table>
<% } %>
```

Handling posts

When a form on a view is posted to a web server running an ASP.NET MVC web application, there are several ways to read and use this post data. One can use request variables, which were also available in ASP.NET Webforms, and even map this request data to a custom object's properties for easy and automatic assignments. Another option is to use action method parameters, which makes handling posts much more transparent and testable.

All of the examples in this topic again make use of the `Contact` class defined earlier. Each example can be created in the `HomeController` class, which can be found in the **Controllers** folder of the example code. Note that the `HomeController` class also features a `List<Contact> Contacts`, in which data can temporarily be stored.

Request variables

ASP.NET Webforms already offered a method to retrieve posted data from a form by using the `HttpRequest` object and its `Form` collection. Because ASP.NET MVC is built on top of ASP.NET Webforms, this behavior is also available for us in the ASP.NET MVC framework.

```
public ActionResult UpdateContact()
{
    int id = 0;
    if (int.TryParse(Request.Form["Id"], out id))
    {
        Contact contact = Contacts.Single(c => c.Id == id);
        contact.Name = Request.Form["Name"];
        contact.Email = Request.Form["Email"];
    }
    return RedirectToAction("Index");
}
```

This code checks the HTTP request's form parameters for a parameter named `Id`. If it can be parsed into an integer, a `Contact` with that ID is retrieved from the database and updated with form values such as name and email address.

Updating objects from request variables

One of the disadvantages of the request variables approach is that a developer using it has to do a lot of type casting (everything is a string), and that the affected object has to be assigned with the data manually.

Each controller class that inherits `System.Web.Mvc.Controller` contains a method named `UpdateModel`, which accepts two parameters: the object that should be updated, and a list of keys that should be mapped from the posted HTML form. For this to work, it is important that your HTML form fields are given the same name as the property of your model: the `Name` property of our `Contact` class should be in an HTML form field named `Name` for it to work.

```
public ActionResult UpdateContact()
{
    int id = 0;
    if (int.TryParse(Request.Form["Id"], out id))
    {
        Contact contact = Contacts.Single(c => c.Id == id);
        UpdateModel(contact, new string[] { "Name", "Email" });
    }

    return RedirectToAction("Index");
}
```

The above code checks the HTTP request's form parameters for a parameter named `Id`. If it can be parsed into an integer, a contact with that ID is retrieved from database and updated by the controller's `UpdateModel` method. This method takes a model, in this case, the `Contact` that has just been retrieved, and a list of form fields corresponding to the model's properties. Only the form fields specified here will be reflected in the model.

Action method parameters

Some developers prefer to simply write an action method on a controller taking some .NET parameters as inputs. The ASP.NET MVC framework will try to map each corresponding HTML field to a method parameter. For example, `<input type="text" id="name" name="name" />` will map to the parameter, `name`.

```
public ActionResult UpdateContact(int id, string name, string email)
{
    Contact contact = Contacts.Single(c => c.Id == id);
    contact.Name = name;
    contact.Email = email;

    return RedirectToAction("Index");
}
```

The ASP.NET MVC framework passes in all form variables as method parameters such as `ID`, `name`, and `email`. These parameters can be used directly to retrieve a contact from the database and update its properties.

One advantage of using this approach is that you do not have to do a lot of type casting as the ASP.NET MVC framework will take care of that for you. This results in much cleaner code, as you can see in the example.

Note that you can easily mix this approach with reading request variables. However, using pure action method parameters allows for easier unit testing and maintainability of your controller actions.

Handling file uploads

Modern web applications often have a file upload form. In classic ASP.NET, these forms were built using the ASP.NET file upload control. In the ASP.NET MVC framework, there is no such thing. When working with file uploads in the ASP.NET MVC framework, one should create an old skool HTML form that accepts file uploads, and then read the uploaded data in a controller action.

Creating an upload form

When creating an upload form, one should always include a special HTML attribute in the form tag, specifying the encoding type as `multipart/form-data`. This directive tells a browser to post the HTML form back to the server in multiple parts: one part for regular form fields, and another for each file.

```
<% using( Html.BeginForm( "Upload", "Home", FormMethod.Post, new{
    enctype="multipart/form-data" } ) )
{ %>
    File 1: <input type="file" name="file1" id="file1" /><br />
    File 2: <input type="file" name="file2" id="file2" />

    <input type="submit" id="upload" value="Upload" />
<% } %>
```

Creating an upload controller action

Handling an upload form is quite easy. The `HttpRequest` object provides property `Files` that contain a set of `HttpPostedFileBase` instances. These contain all sorts of information for a specific file that is being uploaded: content length, content type, and the filename on the user's computer. The `SaveAs()` method allows you to save the uploaded file somewhere on the web server or to a connected network directory.

```
public ActionResult Upload()
{
    StringBuilder info = new StringBuilder();
    foreach (string file in Request.Files)
```

```
{
    HttpPostedFileBase postedFile = Request.Files[file];
    if (postedFile.ContentLength == 0)
        continue;

    /* The following would save the file on the server:
    * string newFileNameOnServer = Path.Combine(
    *     AppDomain.CurrentDomain.BaseDirectory,
    *     Path.GetFileName(postedFile.FileName));
    * postedFile.SaveAs(newFileNameOnServer);
    */

    info.AppendFormat("Uploaded file: {0}\r\n",
                      postedFile.FileName);
}
if (info.Length > 0)
{
    ViewData["Info"] = info.ToString();
}
return View("UploadForm");
}
```

As you can see, the file upload is being passed just like a regular ASP.NET file upload—the `files` property of the `HttpRequest` contains a list of all uploaded files. Each file is checked for size and can be saved permanently on the web server or to some network location.

Using the `ModelBinder` attribute

Action methods can be developed using regular method parameters. In the earlier examples, these parameters were all simple types such as integers, strings, booleans, and so on. Using the ASP.NET MVC `ModelBinder` infrastructure, binding form values to a model becomes quite easy.

All of the examples in this section again make use of the `Contact` class, defined earlier. The examples in this section are based on an ASP.NET MVC web application that can be found in the sample code for this book (`ModelBinderExample`). This sample project contains one controller, `HomeController`, and three views of which we will be using one, `NewContact.aspx`.

Using the default ModelBinder

By default, the ASP.NET MVC framework provides a model binder that allows you to bind form data to complex types.

Consider our Contact class:

```
public class Contact
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

This class will be bound to the following form, which enables a user to add a new Contact to the application:

```
<h2>New contact</h2>
<h3><%=Html.Encode(ViewData["title"] ?? "")%></h3>
<% using (Html.BeginForm("NewContact", "Home", FormMethod.Post)) { %>
    <table border="0" cellpadding="2" cellspacing="0">
        <tr>
            <td>Name:</td>
            <td>
                <%=Html.TextBox("Name",Model.Name ?? "")%>
            </td>
        </tr>
        <tr>
            <td>Email:</td>
            <td>
                <%=Html.TextBox("Email",Model.Email ?? "")%>
            </td>
        </tr>
        <tr>
            <td>&nbsp;</td>
            <td>
                <input type="submit" id="send" value="Add contact" />
            </td>
        </tr>
    </table>
<% } %>
```

When the `NewContact` action method is called, form values are inserted in the `form` parameter of the action method. This parameter is of the type `FormCollection`, which implements the `IValueProvider` interface. The `IValueProvider` interface can be passed to the `UpdateModel` (or `TryUpdateModel`) method of the controller. This method will use the `IValueProvider` instance to populate the model properties.

```
[AcceptVerbs("POST")]
public ActionResult NewContact(FormCollection form)
{
    // Create Contact
    Contact contact = new Contact();
    UpdateModel(contact, new string[] { "Name", "Email" },
    form.ToValueProvider());
    // We should be saving the contact here...
    ViewData["title"] = "Success!";
    // Done!
    return View(contact);
}
```

The above action method accepts a `FormCollection` parameter and uses it to populate a new `Contact` instance (only the properties `Name` and `Email` are populated).

The action method can also be rewritten to simply accept a `Contact` instance:

```
[AcceptVerbs("POST")]
public ActionResult NewContact(Contact contact)
{
    // We should be saving the contact here...
    ViewData["title"] = "Success!";
    // Done!
    return View(contact);
}
```

One can also implement this action method using the `[Bind]` attribute, which allows more control over the model binding behavior:

```
[AcceptVerbs("POST")]
public ActionResult NewContact([Bind(Prefix = "", Include =
"Name,Email")] Contact contact)
{
    // We should be saving the contact here...
    ViewData["title"] = "Success!";
    // Done!
    return View(contact);
}
```

By specifying the `[Bind]` attribute's `Prefix` property, form elements can be mapped using a prefix. For example, if you prefer form fields with the name `cont.Name`, the prefix can be set to `cont`. The `Include` property specifies which properties of the model should be mapped. Optionally, properties can also be excluded by using the `Exclude` property.

Always be careful when using the model binder infrastructure; make sure that you do not allow properties to be mapped if they should not be mapped (for example, IDs, references to other objects, and so on).

Creating a custom ModelBinder

When required, a method parameter can be a complex type such as a `Contact` with `Id`, `Name` and `Email` properties. In almost any situation, the default model binder that is registered in the ASP.NET MVC framework should be able to map a form post to a complex type. There might be situations when you would want to perform model binding yourself, or when the ASP.NET MVC default model binder is not sufficient. In this case, you are required to add a `ModelBinder` attribute or register the `ModelBinder` using the `ModelBinders.Binders.Add()` method.

Take the previously-created `NewContact` action method:

```
[AcceptVerbs("POST")]
public ActionResult NewContact(Contact contact)
{
    // We should be saving the contact here...

    ViewData["title"] = "Succes!";
    // Done!
    return View(contact);
}
```

We are going to create a custom model binder that can determine the contact's name when only the email address is specified. For example, if the name is not filled out in the HTML form and the email address is `maarten@maartenballiauw.be`, we are going to populate the `Name` property with `maarten`.

The `Contact` class should be decorated with a `ModelBinder` attribute so that a form post can be converted to a complex type:

```
[ModelBinder(typeof(ContactBinder))]
public class Contact
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

Alternatively, this `ModelBinder` can be registered globally when the web application is started. It is also possible to register a default model binder that will be used when a specific binder type is not found. For example, the `DefaultModelBinder` class can be explicitly registered as a default model binder that will try to map form key-value pairs to object properties. Note that this code example is just for reference: `DefaultModelBinder` is always the default model binder and does not have to be explicitly registered.

```
void Application_Start()
{
    ModelBinders.Binders[typeof(Contact)] = new ContactBinder();
    ModelBinders.DefaultBinder = new DefaultModelBinder();

    RegisterRoutes(RouteTable.Routes);
}
```

In the web application's `Application_Start` event handler, a model binder can be registered for any complex type in the application. An example of this is the `ContactBinder` type, which implements `IModelBinder`, and provides the functionality that we want to have implemented:

```
public class ContactBinder : IModelBinder
{
    #region IModelBinder Members
    public object BindModel(ControllerContext controllerContext,
                           ModelBindingContext bindingContext)
    {
        if (bindingContext.ModelType == typeof(Contact))
        {
            // Get values from form
            string nameFromForm = bindingContext.ValueProvider["Name"]
                                   .AttemptedValue;
            string emailFromForm = bindingContext.ValueProvider
                                   ["Email"].AttemptedValue;
            if (string.IsNullOrEmpty(nameFromForm) &&
                !string.IsNullOrEmpty(emailFromForm))
            {
                nameFromForm = emailFromForm.Substring(0,
                                                         emailFromForm.IndexOf("@"));
            }

            // Create the Contact
            Contact contact = new Contact();

            // Assign properties
            contact.Name = nameFromForm;
        }
    }
}
```

```

        contact.Email = emailFromForm;
        // Create and return result
        return contact;
    }
    return null;
}
#endregion
}

```

The above code implements the `IMoDelBinder` interface and accepts a `ModelBindingContext` parameter. This method returns an object that should be populated with the data from the `ModelBindingContext`.

The ASP.NET MVC framework offers some default `IMoDelBinder` implementations:

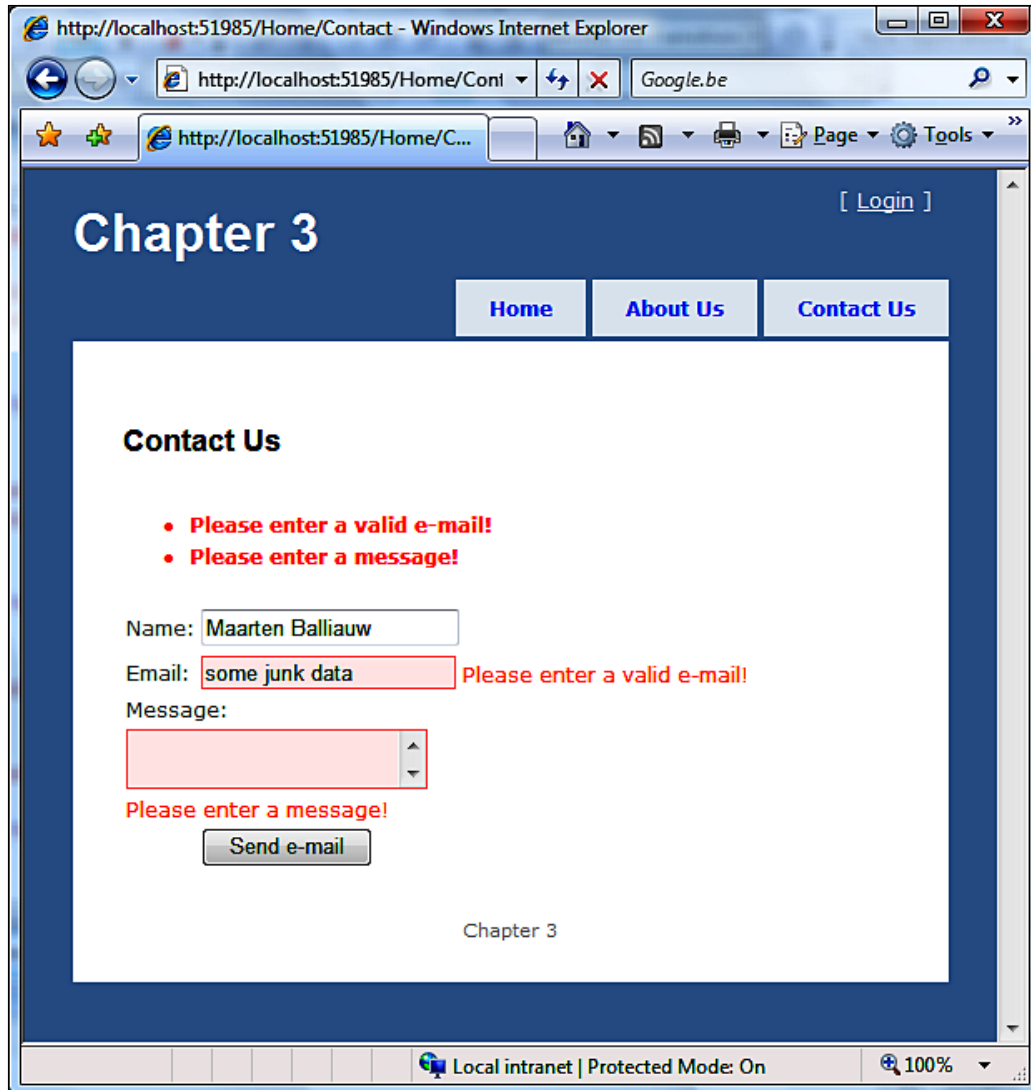
Class name	Description
<code>DefaultModelBinder</code>	Default model binder, used for all simple types and arrays. It is possible to use this as the base class for custom model binders and override the <code>ConvertType()</code> method. It also allows the binding of a parameter to a complex type, for example, a <code>Contact</code> class with a property of type <code>Address</code>. The HTML form field names can be <code>Name</code>, and <code>Email</code> for the <code>Contact</code> properties and <code>Address.Street</code>, <code>Address.City</code>, and such, for the <code>Contact.Address</code> properties.
<code>FormCollectionModelBinder</code>	Allows binding to a HTML form collection.

Validating data

One of the common requirements when creating an application is to validate data and inform the user if something goes wrong. ASP.NET Webforms offered validation controls and an optional validation summary. When invalid values were entered in a specific control, an error message was displayed. The validation summary would show all of the generated error messages.

The ASP.NET MVC framework offers the same possibilities but in a different manner. The `HtmlHelper` class provides two methods: `ValidationMessage` and `ValidationSummary`. The first one will display a message for a specific control, while the latter will render a complete summary of all generated validation errors.

The examples in this topic are based on an ASP.NET MVC web application, which can be found in the sample code for this book (ValidationExample). This sample project contains one controller, `HomeController`, and three views, of which we will be using one, `Contact.aspx`.



Of course, for validation to work, the ASP.NET MVC validation framework has to be informed of possible errors. You can easily do this in your controller action method. The following action method accepts a name, email, and message from an HTML form post.

```
public ActionResult Contact(string name, string email,
                           string message)
{
    // Add data to view
    ViewData["name"] = name;
    ViewData["email"] = email;
    ViewData["message"] = message;

    // Validation
    if (string.IsNullOrEmpty(name))
        ViewData.ModelState.AddModelError("name",
            "Please enter your name!");
    if (string.IsNullOrEmpty(email))
        ViewData.ModelState.AddModelError("email",
            "Please enter your e-mail!");
    if (!string.IsNullOrEmpty(email) && !email.Contains("@"))
        ViewData.ModelState.AddModelError("email",
            "Please enter a valid e-mail!");
    if (string.IsNullOrEmpty(message))
        ViewData.ModelState.AddModelError("message",
            "Please enter a message!");

    // Send e-mail?
    if (ViewData.ModelState.IsValid)
    {
        // send email...
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}
```

In this code snippet, validation is performed on incoming parameters. Whenever a parameter is missing, the `ModelState` collection of `ViewData` is updated with a model error. The `ViewData.ModelState` collection holds a list of all possible error messages. Errors can easily be added using the `ViewData.ModelState.AddModelError` method, which takes a key (that can be used in the view) for the error, a model property name, and an error message.

An alternative to the above approach would be to delegate validation to the model. Whenever the model is updated with invalid data, it should throw an exception. These exceptions should be caught and reflected in the `ViewData.ModelState` collection.

After adding the error messages, the state of the model can be retrieved from the `ViewData.ModelState.IsValid` property. If no errors were added, this property will return `true`, otherwise it will return `false`.

In the view, the `HtmlHelper.ValidationSummary` and `HtmlHelper.ValidationMessage` methods can be used to interact with this `ViewData.ModelState` collection.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.
Master" AutoEventWireup="true" CodeBehind="Contact.aspx.cs" Inherits="
ValidationExample.Views.Home.Contact" %>

<asp:Content ID="Content1"
    ContentPlaceHolderID="MainContent"
    runat="server">
    <h2>Contact Us</h2>

    <p><%=Html.ValidationSummary() %></p>

    <% using (Html.BeginForm(Model.PostAction, "Home", new { id
    =Model.Contact.Id }, FormMethod.Post)) { %>
        <table border="0" cellpadding="2" cellspacing="0">
            <tr>
                <td>Name:</td>
                <td>
                    <%=Html.TextBox("name", ViewData["name"] ?? "") %>
                    <%=Html.ValidationMessage("name") %>
                </td>
            </tr>
            <!-- ... more fields ... -->
        </table>
    <% } %>
</asp:Content>
```

The `HtmlHelper.ValidationSummary` method will render a summary of all of the validation errors in the `ViewData.ModelState` collection. The `HtmlHelper.ValidationMessage` renders a single validation error specific to a given property name.



In most cases, the validation message that is displayed is based on the validation message that is found in the `ViewData.ModelState` collection. Although this is clear information to show in a `ValidationSummary`, it might be too much information for a `ValidationMessage`. On most web sites, a single error is often displayed as an asterisk (*), next to the invalid field. By setting the second parameter of `ValidationMessage` to an asterisk (*), there will only be a short notification of the error next to the field, while the `ValidationSummary` will still displays the full error message.

The following line of code will show an error as a red asterisk (*) next to the name field:

```
<%=Html.ValidationMessage("name", " *") %>
```

Summary

In this chapter, we have learned the different methods of creating an HTML form, which can be used to post data to an action method by making use of HTML and the `HtmlHelper` class. We have also seen how to read values from a form post in an action method by using request variables and action method parameters. We have also seen how to handle file uploads via an action method.

Another thing that we have seen is the model binder infrastructure provided by the ASP.NET MVC framework. We have also implemented a custom `IMoDelBinder`.

In this chapter, we have also learned how form data can be validated, and if necessary, feedback can be provided to the user of our web application.

4

Components in the ASP.NET MVC Framework

This chapter describes the components that build the ASP.NET MVC framework, from the request life cycle to the components in detail. Each step in the ASP.NET MVC request flow is described and related to the full extent. A set of extension points required to customize the ASP.NET MVC request life cycle is also discussed.

In addition to the ASP.NET MVC request life cycle, other concepts including the model, the view, and the controller are also explained in depth in this chapter. The advanced aspects of designing an ASP.NET MVC application are also covered.

You will learn the following in this chapter:

- What the components that build the ASP.NET MVC framework are
- The ASP.NET MVC request life cycle, in depth
- Possible extension points to the ASP.NET MVC request life cycle; where custom steps can be inserted
- The model, in depth, and how easy validation capabilities can be provided
- The controller, in depth, and what action method attributes are
- The view, in depth, and what master pages and partial views are
- What action filters are, and how they can be used when designing an ASP.NET MVC application

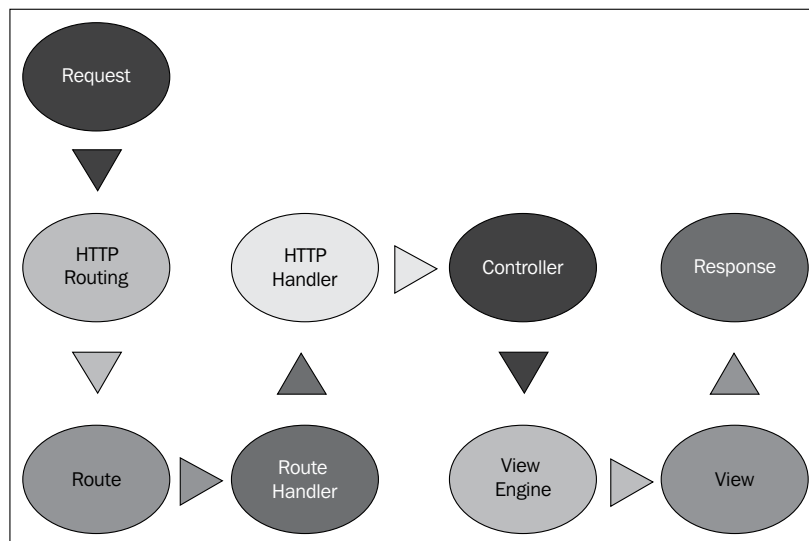
The ASP.NET MVC request life cycle

When an ASP.NET MVC web application request is made, some steps are executed in order to render a response. These steps are also known as the ASP.NET MVC request flow.

Between the HTTP request and the HTTP response, there are eight main steps that occur:

1. The RouteTable is created.
2. The UrlRoutingModule intercepts the request.
3. The routing engine determines the route.
4. The route handler creates the associated IHttpHandler.
5. The IHttpHandler determines the controller.
6. The controller executes.
7. A ViewEngine is created.
8. The view is rendered.

These steps are illustrated in the following diagram:



The RouteTable is created

This step occurs *only* when an ASP.NET MVC application starts, or after the web server's application pool is restarted.

In normal situations, each request to an ASP.NET page corresponds to a page on the disk. For example, requesting `http://www.example.com/Products.aspx` corresponds to the `Products.aspx` page on the web server's disk. This `Products.aspx` page is actually a class that is instantiated and processed by an ASP.NET `IHttpHandler`. An `IHttpHandler` contains a `ProcessRequest` method, which is responsible for rendering the page's output back to the browser.

An ASP.NET MVC web application acts differently. Each incoming request is mapped to a route that determines the correct controller instance that will process the actual request. These routes are defined in a route table that is created every time the web application is started. The `Global.asax` file, which contains code for handling application-level and session-level events, contains an event handler `Application_Start`, which is called on application startup.



Routing is explained in detail in Chapter 5, *Routing*.

The `UrlRoutingModule` intercepts the request

Each incoming HTTP request is intercepted by the `UrlRoutingModule`, which provides the current `HttpContext` to the routing engine.

All data related to the current request is available in an `HttpContext` instance. The `UrlRoutingModule` delegates control to the routing engine and provides the current `HttpContext` data to the routing engine.

The `UrlRoutingModule` implements `IHttpModule`, and is registered in your ASP.NET MVC web application's `web.config` file.

The routing engine determines the route

The routing engine, `UrlRoutingModule`, determines the route handler based on the current `HttpContext`. It locates a matching route in the route table and makes sure that a route handler is created in the form of an `IRouteHandler` instance.

The route handler creates the associated `IHttpHandler`

In the route table, each route is associated with an `IHttpHandler`. This `IHttpHandler` is ultimately responsible for creating the correct controller based on data in the `HttpContext`. The `IHttpHandler` is instantiated by the currently-active `IRouteHandler`.

The `IHttpHandler` determines the controller

For most ASP.NET MVC requests, the `IHttpHandler` will be `System.Web.Mvc.MvcHandler`. This class creates an instance of the controller that is associated with the route based on incoming `HttpContext` and URL parameters.

The controller is created from the `CreateController` method of `ControllerFactory`. A `ControllerContext` is populated from all known contextual data at this point, and passed into the controller's `Execute` method, which triggers the controller's logic.

The controller executes

All controller logic is called, and requested actions are executed. This is the point when interaction with the model occurs. When the controller's logic has been executed, an `ActionResult` is returned. An `ActionResult` instance can, for example, trigger the rendering of a view. When this occurs, a `ViewEngine` is created and instructed to handle further processing.

A ViewEngine is created

The `ViewEngine` instance will create an instance of `IView`, which is returned in a `ViewEngineResult` instance. This `IView` instance will be responsible for the actual rendering of the view.

The view is rendered

The `ViewEngineResult` instance that was returned by the `ViewEngine` contains an `IView` instance. This `IView` instance compiles the requested view and populates its data when its `Render` method is called.

Extensibility

When working with the ASP.NET MVC framework, you must be aware that there are several extensibility points. Of course, you can implement your own controller logic and view logic, but whenever it's required, some custom extensions can be used. Almost all of the steps in the ASP.NET MVC request flow can be extended or customized.

Route objects

When building the route table, you can call the `Add` method of `RouteCollection` to add new `Route` objects. Each `Route` object is based on the `RouteBase` abstract class, which is required by the `RouteCollection`. You can implement your own `Route` objects that inherit from the `RouteBase` class, if needed. An example of this would be a custom `RouteBase` implementation that ignores requests for a particular file extension.

MvcRouteHandler

The route table maps certain URL patterns to an `MvcRouteHandler`. A route can be mapped to any class that implements the `IRouteHandler` interface. The `GetHttpHandler` of this class should return a valid `IHttpHandler`.

An alternative to this is to override the `GetHttpHandler` method of `MvcRouteHandler`.

ControllerFactory

By default, controllers are instantiated by the `DefaultControllerFactory` class. You can create your own controller factory by implementing the `IControllerFactory` interface. The `IControllerFactory` instantiates a controller for a given controller name and `RequestContext`. Note that your custom `IControllerFactory` implementation should be registered by calling the `System.Web.Mvc.ControllerBuilder.Current.SetControllerFactory()` method when your application starts. This can be done in the `Application_Start` method of `Global.asax`.

Controller

Each controller that you create using the Visual Studio templates will inherit the `Controller` class. You can create your own controller or base class by implementing the `IController` interface. You are only required to implement a single method: `Execute`. This method will execute the controller logic based on a given `ControllerContext`.

ViewEngine

It is possible to create a custom `ViewEngine`. The default `ViewEngine` is determined from the static class `ViewEngines.DefaultViewEngine`. The static collection `ViewEngines.Engines` can be extended with your own implementation of the `IViewEngine` interface or the `ViewEngineBase` abstract class. You should implement the `RenderView` and `RenderPartial` method, given a `ViewContext` instance.

A `ViewEngine` also maps view names to actual files on the web server. By default, views are located inside the **Views | ControllerName** project folder or the **Views | Shared** folder. You can always customize this by creating your own `ViewEngine` and registering it in the `Application_Start` method of `Global.asax`:

```
protected void Application_Start()
{
    // other code...
    // Add ViewEngine
    ViewEngines.Engines.Add(new MyCustomViewEngine());
}
```

View

By default, the `ViewEngine` will instantiate an `IView` of the type, `WebFormView`. The `WebFormView` class handles ASP.NET MVC's default view based on HTML and ASP.NET markup. On the Internet, there are some other `ViewEngine` and `IView` combinations that feature different kinds of markup in which a view can be created. If you want to, you can also create your own implementation. Chapter 6, *Customizing & Extending the ASP.NET MVC Framework* of this book features an example of this.

The model in depth

In a model-view-controller application, the model is the portion of the application that is responsible for handling business logic. Typically, model objects access data from a persistent store such as a database or an XML file, and perform business logic on that data. Most models are application-specific, as they actually define how interaction with data occurs in a specific situation. The ASP.NET MVC framework offers access to any kind of model that is built in a .NET language: ADO.NET DataSets, DataReader objects, domain objects, object-relational mappers, LINQ to SQL classes, and so on.

When talking about the model, it seems like there's only one class responsible for handling business logic. In reality, you will have several classes in a separate namespace or assembly that are performing business logic. This set of classes is typically not aware of browsers or the presentation layer, for example, the HTML output. Instead, it only knows about data and processes.

Creating a model

A common practice, when developing the model or business logic, is to use a set of domain classes. For example, a task scheduler application would define a `Task` class inside the domain layer:

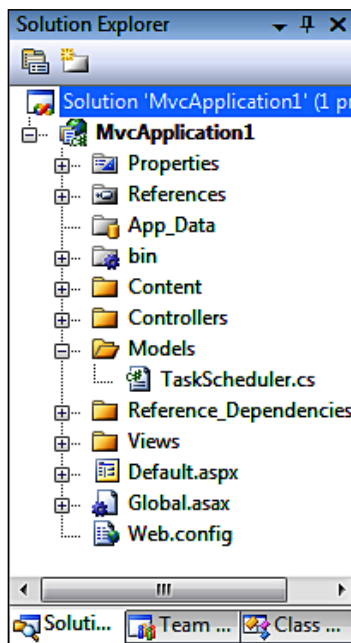
```
public class Task {  
    public int Id { get; set; }  
    public DateTime DueDate { get; set; }  
    public string Title { get; set; }  
    public string Description { get; set; }  
}
```

This class is then used in the business layer:

```
public class TaskScheduler  
{  
    public bool ScheduleTask(DateTime when, string title, string  
        description) { }  
    public bool UpdateTask(Task task) { }  
    public IList<Task> RetrieveAllTasks() { }  
    public IList<Task> RetrieveDueTasks() { }  
}
```

This business class defines how interaction between your application and the underlying data occurs. There's no room for direct database access. Instead, a set of methods is provided that access data and return domain class instances.

Model classes can be placed inside the **Models** folder of your ASP.NET MVC application. Another option is to create a separate assembly. This last option allows you to re-use the model in other applications—for example, a Windows application that provides the same features as your ASP.NET MVC application.



Using the model inside a controller typically consists of instantiating the model class in the controller actions, calling one or more methods on the model, extracting the appropriate data, and displaying it inside a view. This allows for a clean separation of concerns, as the controller is not occupied with business logic. A good practice for passing this information into the view is to embed it into a custom `ViewData` class.

Here's an example that will show all of the tasks that are due by a given date:

```
public class TaskController : Controller
{
    public class DueTasksViewData
    {
        public string Title { get; set; }
        public IList<Task> Tasks { get; set; }
    }
    public ActionResult Index()
    {
        TaskScheduler scheduler = new TaskScheduler();
        DueTasksViewData viewData = new DueTasksViewData
        {
            Title = "Overview of due tasks",
            Tasks = scheduler.RetrieveDueTasks()
        };
        return View("Index", viewData);
    }
}
```

In this example code, a class instance named `DueTasksViewData` is created and populated with a page title and a list of tasks that are due. This `DueTasksViewData` instance is passed to the view, which can use the `Title` and `Tasks` property to display the required data.

Enabling validation on the model

When working with the `UpdateModel` method in a controller, as seen in Chapter 3, *Handling Interactions* of this book, a model object can be updated from the form data that a user posts to the controller. By wrapping this `UpdateModel` call in a try-catch block, one can easily enable validation of the model and provide support for the `ValidationSummary` and `ValidationMessage` methods of `HtmlHelper`.

The following action method accepts a form post and tries to update a contact by using the `UpdateModel` method. If this fails, the exception is caught and the `Contact` is passed to the view. This view then re-renders the edit form, showing the validation errors by using `HtmlHelper.ValidationMessage()`:

```
public ActionResult Edit(int id, FormCollection form)
{
    Contact contact = new Contact();
    try
    {
        UpdateModel(contact, form.ToValueProvider());

        return RedirectToAction("Index");
    }
    catch {
        return View(contact);
    }
}
```

There is one more thing left to do—the validation itself. `UpdateModel` will look for the `IDataErrorInfo` interface for our model, and if that is found, validation will occur automatically. `IDataErrorInfo` is an interface definition that is found in `System.ComponentModel`. It provides the functionality to offer custom error information that a user interface can bind to. Here's a `Contact` model that implements `IDataErrorInfo`:

```
public class Contact : IDataErrorInfo
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    #region IDataErrorInfo Members
    public string Error
    {
        get { return ""; }
    }
    public string this[string columnName]
    {
        get
        {
            switch (columnName.ToUpperInvariant())
            {
                case "NAME":
                    if (string.IsNullOrEmpty(Name))
                        return "Please provide a valid name.";
                    break;
                case "EMAIL":
                    if (string.IsNullOrEmpty(Email))
                        return "Please provide a valid email.";
                    break;
            }
        }
    }
}
```

```
        return "";  
    }  
}  
#endregion  
}
```

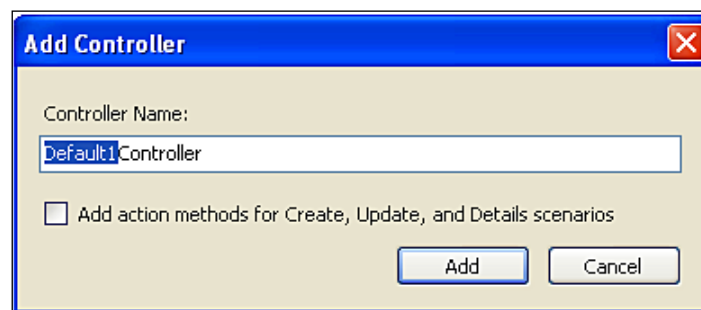
The `Error` property that is defined on `IDataErrorInfo` returns a string containing any error that is "global" for the `Contact` object. The `this[string columnName]` indexer that is defined on `IDataErrorInfo` is used to retrieve error messages for a specific property. For example, if the `Name` is empty, the `this["Name"]` method will return an error stating that the name is not specified. The ASP.NET MVC model binder will then reflect this in an `Exception`, and pass it to a view, where it can be used by, for example, `ValidationSummary` or `HtmlHelper`.

The controller in depth

In an MVC application, the controller is the portion of the application that is responsible for communicating the data from the model to a view. Typically, the controller does not contain any business logic, but instead contains the logic required to communicate with the business layer and the view.

Creating a controller

Creating a controller in Visual Studio 2008 can be done by adding a new item to the project, selecting the **MVC Controller Class** template, which is located under the **Web | MVC** category. Another option is to select the **Add controller...** option when right-clicking on the solution. This menu item shows a form that can be used to specify some details for the controller class to be created:



In this window, you can specify the name of the controller class that should be created. You can also specify if some default action methods (**Create, Update, and Details**) should be created in order to speed up development.

Rendering data on the response

In an ASP.NET MVC web application, the controller is a class that derives from `System.Web.Mvc.Controller` or implements `System.Web.Mvc.IController`. A controller contains one or more action methods, which are basically methods that sit in between the application user's browser and the model. A controller action will, for example, retrieve a list of due tasks from the model and return an `ActionResult` instance. This in turn will (again, for example) render an associated view based on the retrieved data. Here's a list of possible `ActionResult` types:

Type	Description
<code>ViewResult</code>	Renders a specified view to the response stream
<code>PartialViewResult</code>	Renders a specified partial view to the response stream
<code>EmptyResult</code>	Basically does nothing; an empty response is given
<code>RedirectResult</code>	Performs an HTTP redirection to a specified URL
<code>RedirectToRouteResult</code>	Performs an HTTP redirection to a URL that is determined by the routing engine, based on given route data
<code>JsonResult</code>	Serializes a given <code>ViewData</code> object to JSON format
<code>JavaScriptResult</code>	Returns a piece of JavaScript code that can be executed on the client
<code>ContentResult</code>	Writes content to the response stream without requiring a view
<code>FileContentResult</code>	Returns a file to the client
<code>FileStreamResult</code>	Returns a file to the client, which is provided by a <code>Stream</code>
<code>FilePathResult</code>	Returns a file to the client

These `ActionResult` types can be generated by using some helper methods in a class that derives from `System.Web.Mvc.Controller`:

Method	Description
<code>View</code>	Returns a <code>ViewResult</code> instance
<code>PartialView</code>	Returns a <code>PartialViewResult</code> instance
<code>Redirect</code>	Returns a <code>RedirectResult</code> instance
<code>RedirectToAction</code>	Returns a <code>RedirectToRouteResult</code> , based on a given action
<code>RedirectToRoute</code>	Returns a <code>RedirectToRouteResult</code> , based on given route data
<code>Json</code>	Returns a <code>JsonResult</code> instance
<code>JavaScript</code>	Returns a <code>JavaScriptResult</code> instance
<code>Content</code>	Returns a <code>ContentResult</code> instance
<code>File</code>	Returns a <code>FileContentResult</code> , <code>FileStreamResult</code> or <code>FilePathResult</code> based on the input

Here's an example of a controller that has an `Index` action, which redirects a user to the `Show` action of the same controller:

```
using System.Web.Mvc;
namespace MvcApplication1.Controllers
{
    public class TaskController : Controller
    {
        public ActionResult Index()
        {
            return RedirectToAction("Show");
        }
        public ActionResult Show()
        {
            return View("Show");
        }
    }
}
```

Reading data from the request

There are situations where an action method relies on the data being passed—for example, when a form is posted. If your controller defines a `Show` action that requires a specific product ID, you can retrieve this ID in several ways.

One way of retrieving data for a controller action is by using the `HttpRequest` parameters. The following code will read the ID parameter from a URL, that is, from a URL in the form `http://www.example.com/Products/Show?id=5`:

```
public ActionResult Show()
{
    int id = Convert.ToInt32(Request["id"]);
    var viewData = RetrieveData(id);
    return View("Show", viewData);
}
```

A second way of retrieving data for a controller action is to use a method parameter. This method is much more flexible, as it will read the ID parameter from URLs such as `http://www.example.com/Products/Show?id=5`, `http://www.example.com/Products/Show/5`, and eventually from a form post:

```
public ActionResult Show(int id)
{
    var viewData = RetrieveData(id);
    return View("Show", viewData);
}
```

In this last method, the routing engine will take care of the parameter mapping. If the parameter type is not correct for the data being passed in, an exception will be thrown by the routing engine. Also, if you defined constraints on the route, you can deny specific requests if the ID does not match a specific regular expression pattern.



We have already seen more on reading data from the request in Chapter 3, *Handling Interactions*.

Action method selection

The ASP.NET MVC framework uses route data to map the request to the correct action method. By default, the `ControllerActionInvoker` class will use reflection to find a public method on the controller that has the same name as the action in the route data dictionary.

When a method is marked with the `ActionNameAttribute`, the `ControllerActionInvoker` will use the action name specified in this attribute, instead of the method name. The following example will not map the URL `/Home/SomeMethodName` to this action method. It will be mapped to a URL in the form of `/Home/Display`.

```
[ActionName("Display")]
public ActionResult SomeMethodName(int id)
{
    return View();
}
```

Note that `return View()` will look for a view named `Display.aspx`, and not for `SomeMethodName.aspx`!

When a method is marked with the `ActionSelectionAttribute`, the `ControllerActionInvoker` will first verify if an action method can really be used to serve a specific request. One can implement this abstract base class and use it to specify in detail which method should handle which request, based on the current `ControllerContext`.

```
public abstract class ActionSelectionAttribute : Attribute
{
    public abstract bool IsValidForRequest(ControllerContext
        controllerContext, MethodInfo methodInfo);
}
```

Another attribute that can be used is the `AcceptVerbsAttribute`. When a method is marked with the `AcceptVerbsAttribute`, which is an implementation of the `ActionSelectionAttribute`, the method will only be used as the action method for a request if the HTTP verb is valid. For example, one can specify one action method to handle only GET requests, while another would handle POST requests. The following code snippet features two action methods named `Edit`, of which the first one will handle all HTTP GET requests, while the second one will handle form posts:

```
public class HomeController : Controller
{
    [AcceptVerbs(HttpVerbs.Get)]
    public ActionResult Edit()
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(FormCollection form)
    {
        // some code to handle edit...
        return View();
    }
}
```

Handling unknown controller actions

The situation might occur where a hyperlink is not correct, or a user requests an action method that does not exist. By default, the ASP.NET MVC framework will render an error page, notifying the user of a 404 error: `page not found`.

If the user requests, say, `http://www.mysite.com/Home/Products`, it is quite clear that the user wants to view a list of products or something similar. In this case, a classic error page might not be the best thing to display to the user. Instead, notifying the user that he probably meant to navigate to `http://www.mysite.com/Products` might be a better alternative. For this behavior, the `HandleUnknownAction` method of the controller base class can be overridden.

The examples in this topic are based on an ASP.NET MVC web application that can be found in the sample code of this book (`HandlingUnknownActionsExample`).

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
    }
```

```

        // ...
    }

    public ActionResult About()
    {
        // ...
    }

    protected override void HandleUnknownAction(string actionName)
    {
        ViewData["Title"] = "Error 404 - Page not found";
        ViewData["Message"] = "Page not found";

        ViewData["RequestedAction"] = actionName;
        ViewData["Controller"] = "Home";
        ViewData["AlternativeActions"] = new string[] { "Index",
                                                        "About" };

        Response.StatusCode = 404;
        View("404").ExecuteResult(ControllerContext);
    }
}

```

The above `HandleUnknownAction` will be triggered whenever an unknown action is called. The `HandleUnknownAction` method is passed the requested action name, which can be used to determine alternative options. In this method, a 404 status code is set, and a 404 view is rendered. Note that this should be done manually by calling the `ExecuteResult` method of `ViewResult`. Otherwise, the ASP.NET MVC framework will create its own error message.

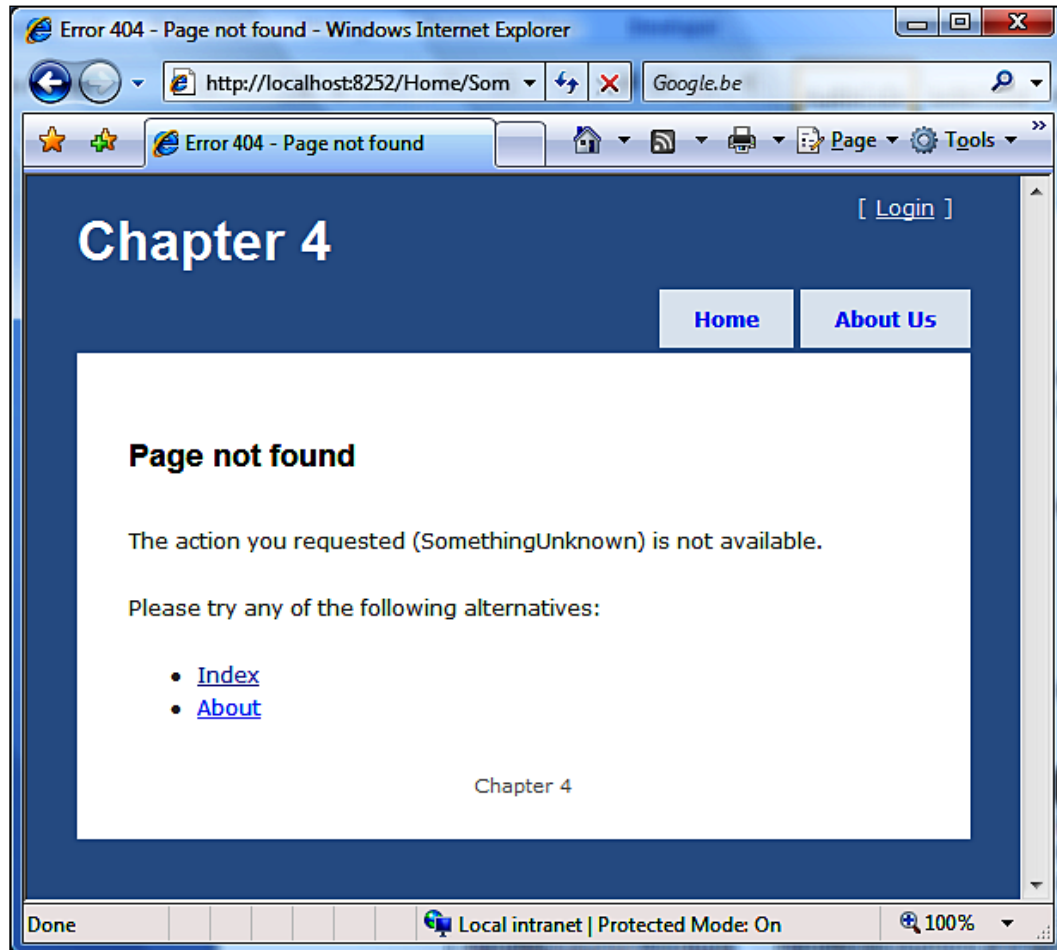
```

<h2><%= Html.Encode(ViewData["Message"]) %></h2>
<p>
    The action you requested (<%=Html.Encode(ViewData[
        "RequestedAction"])%>) is not available.
</p>
<p>
    Please try any of the following alternatives:
    <ul>
        <%
            string[] alternatives = ViewData["AlternativeActions"] as
                string[];
            foreach (string alternative in alternatives)
            {
                %><li><%=Html.ActionLink(alternative, alternative,
                    ViewData["Controller"].ToString())%></li><%
            }
        %>
    </ul>
</p>

```

The 404 view will simply render all ViewData elements that were passed in, along with a list of action links to the alternative actions that were set in the `HandleUnknownAction` method on the controller.

Here's how a request to `http://www.mysite.com/Home/SomethingUnknown` would look like:



Action method attributes

A number of action method attributes can be applied to the controller or to the action method itself.

Some examples of action method attributes can be found in the sample code (ActionMethodAttributesExample). In the `HomeController`, each action method attribute is demonstrated with comments.

Attribute	Description
<code>NonActionAttribute</code>	Hides the method for the <code>ControllerActionInvoker</code> and will prohibit the method from being called as an action method. Use this attribute on helper methods that are located in a controller class. The following action method will never be available as an action method, that is, <code>http://example.com/Home/NonAction</code> will never be mapped by the routing engine: <pre>[NonAction] public ActionResult NonAction() { return View("Home"); }</pre>
<code>HandleErrorAttribute</code>	Handles exceptions and optionally renders a different view when an exception occurs. Note that this attribute is ignored when debugging. The following action method will render <code>About.aspx</code> when an <code>InsufficientMemoryException</code> occurs: <pre>[HandleError(ExceptionType=typeof(InsufficientMemoryException), View="About")] public ActionResult HandleError() { throw new InsufficientMemoryException(); return View("Home"); }</pre>
<code>AuthorizeAttribute</code>	Secures an action method; verifies if the current user is authenticated and optionally verifies the user name or role of the user. (More on this attribute in Chapter 7.)
<code>OutputCacheAttribute</code>	Caches the output of an action method using the ASP.NET output cache. (More on this attribute in Chapter 7)

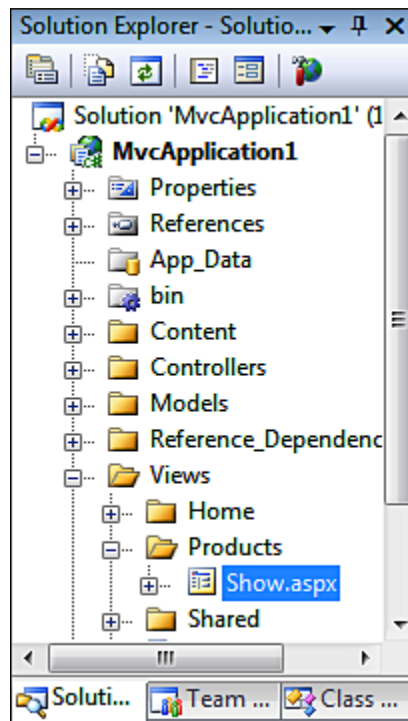
Attribute	Description
AcceptVerbsAttribute	Marks the action method as callable only when the HTTP verb is specified in the attribute. For example, an action method can be marked to handle only POST requests. The following action method will only render a view when an HTTP POST is being made: <pre>[AcceptVerbs(HttpVerbs.Post)] public ActionResult PostOnly() { return View("Home"); }</pre>
ActionNameAttribute	Marks the action method with another name for the ControllerActionInvoker. The following action method will not be known as xyz123, but will instead be known as ActionName: <pre>[ActionName("ActionName")] public ActionResult xyz123() { return View("Home"); }</pre>
ValidateAntiForgeryToken	Checks to see if the current request contains a valid anti-forgery token that was generated by the <code>HtmlHelper.AntiForgeryToken()</code> helper method.

The view in depth

In an MVC application, the view is the portion of the application that is responsible for displaying model data that has been received from the controller, in a presentation layer. When developing a web application using the ASP.NET MVC framework, most of your views will render HTML output to a client's browser, based on model data.

Location of views

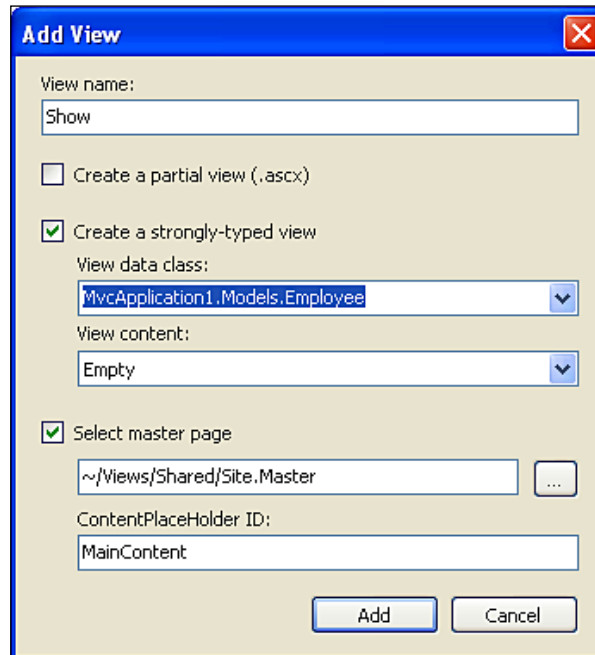
In an ASP.NET MVC web application, views are typically stored in the **Views | [Controller]** folder. For example, a controller named `ProductController` will have its associated view, `Show`, located in **Views | Product | Show.aspx**.



If a specific view is associated with multiple controllers, you can also store it in the **Views | Shared | Show.aspx** folder. This folder will typically contain a master page, but might also contain a specific view as well.

Creating a view

Creating a view in Visual Studio 2008 can be done by adding a new item to the project, selecting the **MVC View Content Page** template, located in the **Web | MVC** category. Another option is to select the **Add view...** option when right-clicking an action method. This menu item shows a form that can be used to specify some details for the view that is to be created:



The screenshot shows the 'Add View' dialog box with the following fields and options:

- View name:** A text box containing 'Show'.
- ☐ Create a partial view (.ascx)
- ☒ Create a strongly-typed view
- View data class:** A dropdown menu showing 'MvcApplication1.Models.Employee'.
- View content:** A dropdown menu showing 'Empty'.
- ☒ Select master page
- Master page:** A text box containing '~\Views\Shared\Site.Master' and a browse button (...).
- ContentPlaceHolder ID:** A text box containing 'MainContent'.
- Buttons:** 'Add' and 'Cancel' at the bottom right.

In this window, you can specify the name of the view that should be created. You can also specify whether it is a partial view, and whether it should be strong-typed or not. The master page that should be used can also be specified in this window.

When creating a strong-typed view, a view content template can be selected. For example, this dialog can generate a view that renders a list, a detail or an edit form.

These templates are T4 templates, and are located in `C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC\CodeTemplates` and can be customised if needed.



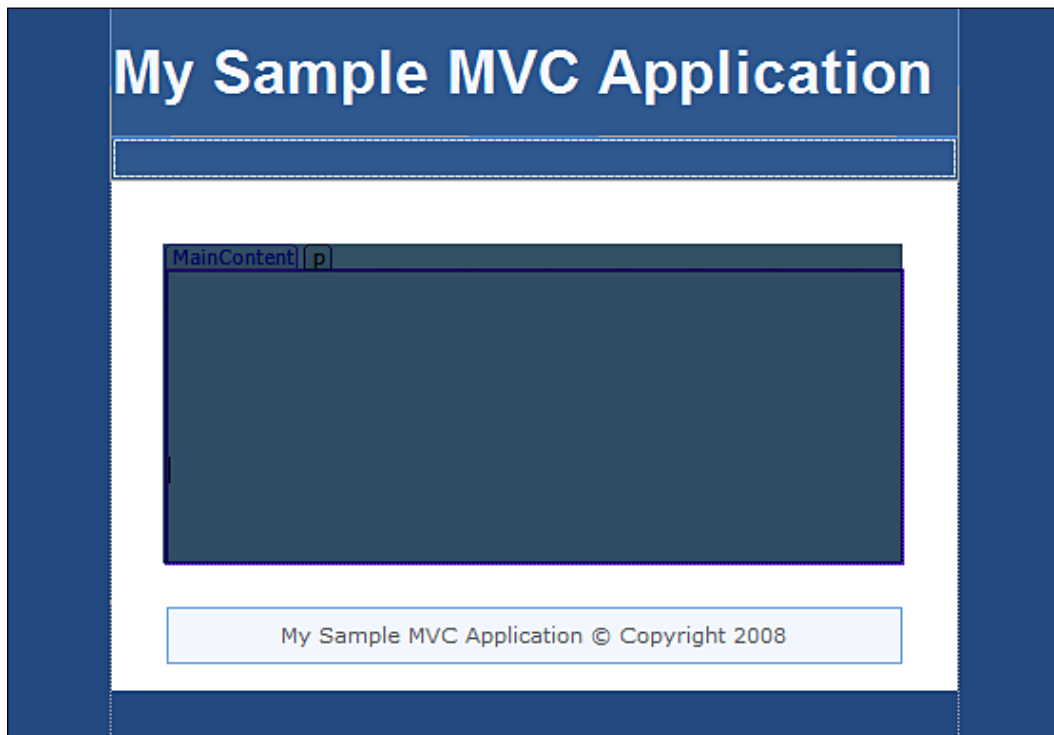
T4 (or Text Template Transformation Toolkit) templates are instructions for Visual Studio's built-in code generator, and can be used for generating code in an application. The view templates are generated using T4, making it possible to create a view that contains all of the information from a model that can be customized later.

A tutorial on creating T4 templates can be found on <http://www.hanselman.com/blog/T4TextTemplateTransformationToolkitCodeGenerationBestKeptVisualStudioSecret.aspx>.

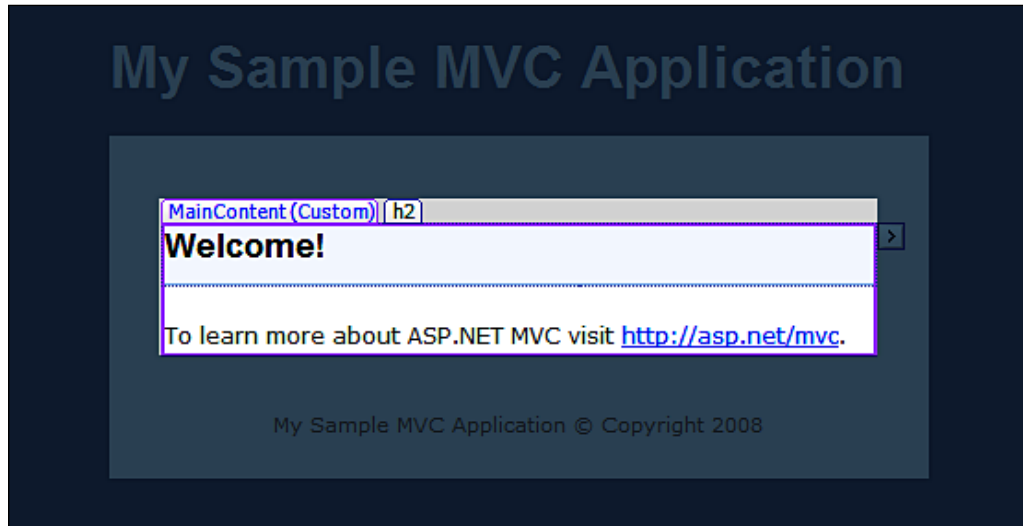
Master pages

In ASP.NET Webforms, you can use a master page to define a global page structure and style for a web application. Afterwards, a content page can be added that will contain only specific contents.

The ASP.NET MVC framework provides support for master pages and content pages. Master pages are stored in the **Views | Shared** folder.



The following screenshot displays the content page. It populates the ContentPlaceHolder of **MainContent** that is defined in the master page:



View markup

The ASP.NET MVC framework provides built-in support for using "regular" .aspx pages as a view. There's only one big difference to note: there's no ViewState or postbacks involved. As a result of this difference, you cannot use all of the controls that you could previously use in ASP.NET Webforms. For example, the ASP.NET grid view relies heavily on ViewState and postbacks and, therefore, cannot be used. In addition, note that the ASP.NET Webforms controls will not always render "clean" HTML code.

As an alternative to ASP.NET Webforms controls, the ASP.NET MVC framework offers inline coding as a valuable alternative if you want full control over the HTML markup that is rendered. Inline code is always placed between `<%` and `%>`. A shortcut for writing out a value to the response stream is `<%=myString%>`, but you can also use the full-blown version: `<% Response.Write(myString); %>` if you prefer to do so.

This topic will use the following Task class as the model:

```
public class Task {  
    public int Id { get; set; }  
    public DateTime DueDate { get; set; }  
    public string Title { get; set; }  
    public string Description { get; set; }  
}
```

Rendering a list of Task objects can be done by using a simple in-line foreach loop:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
AutoEventWireup="true" CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>
<%@ Import Namespace="MvcApplication1.Controllers" %>
<%@ Import Namespace="MvcApplication1.Controllers.TaskController" %>
<%@ Import Namespace="MvcApplication1.Models" %>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
    runat="server">
    <h2><%= Html.Encode(Model.Title) %></h2>

    <ul>
    <% foreach (Task task in Model.Tasks) { %>
        <li>
            <%= Html.ActionLink<TaskController>( c =>
                c.Show(task.Id), "Details" )%>
            <%= Html.Encode(task.Title) %>
        </li>
    <% } %>
    </ul>
</asp:Content>
```

The ASP.NET MVC framework features a class called `HtmlHelper` that provides each view with a standard set of helper methods that provide a programmatic method of creating HTML controls. When executing the application, `HtmlHelper` methods are rendered as plain HTML. Every view page provides an `Html` property, which is an instance of the `HtmlHelper` class. This code snippet also uses the `HtmlHelper` class to encode data and to create a hyperlink to a specific controller action. The `HtmlHelper.ActionLink` method is a helper method that accepts a controller type and utilizes the routing engine to generate a hyperlink to a specific action on that controller. For example, the code in the above snippet will generate the following hyperlink:

```
<a href="/Task/Show/9">Details</a>
```

Note that the `HtmlHelper` class is built around extension methods. The `HtmlHelper` class is defined in the `System.Web.Mvc` namespace, and defines some methods that are useful for encoding data and many such things. By default, an ASP.NET MVC application's `Web.config` file contains the following entry:

```
<pages>
    <!-- ... -->
    <namespaces>
        <add namespace="System.Web.Mvc"/>
        <add namespace="System.Web.Mvc.Html"/>
```

```
<add namespace="System.Web.Mvc.Ajax"/>
</namespaces>
<!-- ... -->
</pages>
```

ASP.NET will use this entry to load some namespaces by default when working in a view. The `System.Web.Mvc.Html` namespace contains a lot of extension methods that are available in each view. This approach enables you to remove the `System.Web.Mvc.Html` entry in `Web.config`, and create your own extension methods on `HtmlHelper`, if you want to.

The helper and extension methods defined on the `HtmlHelper` class are described in Chapter 3, *Handling Interactions*.

Partial views

In every ASP.NET MVC application, there are situations where a view contains one or more small, re-usable pieces of views. The ASP.NET MVC framework supports a smaller subset of classic views, namely partial views. Partial views are able to get the model data to display from the parent view's `ViewData`, or through the use of a model that was passed specifically into them.

The key difference with a regular view is that a partial view has no master page support as it is intended for partial rendering. Another difference is that a partial view can be rendered by a view engine other than the parent view. This provides a flexible architecture that, for example, allows some parts of a web site to be rendered by a different view engine, which is easier for web site designers to understand. Partial views are searched for in the same locations as regular views and can be used on a parent view by using the method `RenderPartial` of `HtmlHelper`. Note that there's no equals sign (`=`), as the partial view will render directly to the output stream instead of returning a string.

Note that partial views can be rendered by providing their names and, optionally, the view data that they should render:

```
<% Html.RenderPartial("viewName"); %>
<% Html.RenderPartial("viewName", customer); %>
```

An example of partial views can be found in the sample code for this book (the `PartialViewExample`). In the `Index.aspx` view, a partial view named `CurrentTime.aspx` is rendered:

```
<p>
    <% Html.RenderPartial("CurrentTime"); %>
</p>
```

The `CurrentTime.aspx` view is located in **Views | Shared | CurrentTime.aspx**, and renders the current time to the HTTP response.

```
<%=Html.Encode(DateTime.Now.ToString())%>
```

Action filters

An action filter is an attribute that can be applied to a controller class or an action method. Whenever the controller or action method is called, the action filter is triggered, both before and after execution. Typically, action filters are used for solving problems that can occur in more than one class—the so called cross-cutting concerns. A typical cross-cutting concern is output caching or authentication; both can be required for more than one action method.

The ASP.NET MVC framework offers four types of action filters:

1. `IAuthorizationFilter`
2. `IActionFilter`
3. `IResultFilter`
4. `IExceptionFilter`

Each of the above action filters is assigned a different priority:

`IAuthorizationFilter` is guaranteed to run before `IActionFilter`; `IActionFilter` runs before `IResultFilter`, which runs before `IExceptionFilter`. This approach avoids accidental errors such as caching view output before authentication occurs.

When writing your own action filter, implement the interface that best suits your needs. Each attribute can be prioritized in its category by assigning the `Priority` property on creation. An example of a custom `ActionFilter` can be found later in this chapter.

IAuthorizationFilter

Authorization filters are always the first type of action filters executed, allowing the action method to be cancelled. Instead of the requested action method, you can easily set a different `ActionResult` to be rendered to the response stream.

Beware of security when implementing `IAuthorizationFilter`—in almost all cases, the default authorization filter provides sufficient features for working with authentication and authorization.

IActionFilter

An action filter allows you to run code before and after an action method is called, but before the result of the action method is executed.

The `IActionFilter` interface defines two methods:

- `OnActionExecuting`: Runs before the action method is executed. You can cancel the action method and even replace the `ActionResult` with another one. When the action method is cancelled, no other action filters will be executed. Action filters for which the `OnActionExecuting` was called will also receive a call for `OnActionExecuted`.
- `OnActionExecuted`: Runs after the action method is executed but before the `ActionResult` is executed. This allows you to replace the `ActionResult` with another one.

Exceptions that have been thrown by other action filters or the action method that is being executed can be examined within the filter. It is possible to handle the exception in the action filter, after which the action result will be executed. If the exception is not handled, the action result will not be executed. Optionally, this can also be handled by an `IExceptionFilter` which is described later in this chapter.

IResultFilter

A result filter is very similar to an action filter (see `IActionFilter`). The only difference is that it is executed both before and after the result returned from the action has been executed.

The `IResultFilter` interface defines two methods:

- `OnResultExecuting`: Runs before the action method's action result is executed
- `OnResultExecuted`: Runs after the action method's action result is executed

IExceptionFilter

An exception filter allows you to handle specific exceptions in code. For example, exception filters can be used for logging specific exceptions or for redirecting a user to an error action method that informs them in a friendly way that an error has occurred.

Exception filters are guaranteed to run after all other action filters and result filters have been executed. This approach allows you to replace the action method's action result with a custom `ActionResult`.

Summary

In this chapter, we have learned about the different components that build the ASP.NET MVC framework. We've seen the request life-cycle that processes a request in an ASP.NET MVC web application and its components, in-depth. We have learned each step in the ASP.NET MVC request, and have seen how each of these is related to the full picture. A set of extension points that is used to customize the ASP.NET MVC request lifecycle has also been described.

We have also learned more about the model, view, and controller. We have seen what action methods, master pages, and partial views are. Advanced aspects in designing an ASP.NET MVC application, such as action filters, have also been covered.

5

Routing

Whenever a user requests a URL in an ASP.NET MVC application, the ASP.NET MVC framework uses ASP.NET routing to map this request URL to a controller class and an action method. ASP.NET routing extracts variables in the URL according to a pattern that you define in a routing table, and automatically passes these variables to a controller action method.

You will learn the following in this chapter:

- What ASP.NET routing is
- The difference between ASP.NET routing and URL rewriting
- How the `UrlRoutingModule` fits into the request life cycle
- How routes are mapped to route patterns and controller action methods
- How routes are defined
- What a catch-all parameter is, and what parameter constraints are
- How an ASP.NET MVC application can be combined with an ASP.NET Webforms application

What is ASP.NET routing?

In a regular ASP.NET web application, each URL is mapped to a file on a disk. For example, a request for `http://www.example.com/Products/Show.aspx?id=5` really maps to a file called `Show.aspx` on the web server's disk.

Using ASP.NET routing, you define specific patterns for a URL that map to a certain handler class that will take care of the request. For example, a URL in the form of `http://www.example.com/Products/Show/5` could be matched with a pattern `http://www.example.com/{controller}/{action}/{id}`. The variables between `{` and `}` are populated with the actual values from the request URL, and will map to the `HomeController` and to the `Index` action if no other value can be deduced from the URL.

These URL patterns can also be used to programmatically create URLs that correspond to them. All hyperlinks in your ASP.NET MVC application can be generated this way and can thus easily be managed by maintaining the route table.

ASP.NET routing versus URL rewriting

You may have already heard of URL rewriting. URL rewriting consists of certain regular expression patterns that match an incoming request URL and forward the request to a mapped URL instead. For example, one might create a URL rewriting rule that forwards an incoming request for `http://www.example.com/Products/Beverages` to another URL of `http://www.example.com/Products/Show.aspx?id=5`. URL rewriting alters the request URL and forwards it to another URL.

ASP.NET routing is different. It does not alter the incoming URL. Instead, it extracts specific values from the URL, based on a pattern. These extracted values can be used to determine the handler that will handle the request. You can also use these patterns to generate a URL that will map to a specific handler.

UrlRoutingModule

ASP.NET routing is initiated by an `IHttpModule` named `UrlRoutingModule`. This module is registered inside the ASP.NET MVC application `web.config` file:

```
<?xml version="1.0"?>
<configuration>
  <!-- ... -->

  <system.web>
    <!-- ... -->

    <httpModules>
      <add name="UrlRoutingModule"
          type="System.Web.Routing.UrlRoutingModule,
              System.Web.Routing, Version=0.0.0.0,
              Culture=neutral, PublicKeyToken=31BF3856AD364E35"
        />
    </httpModules>
  </system.web>

  <!-- The system.webServer section is required for running ASP.NET
  AJAX under Internet Information Services 7.0. It is not
  necessary for previous version of IIS. -->
  <system.webServer>
    <!-- ... -->
```

```

    <modules runAllManagedModulesForAllRequests="true">
      <remove name="UrlRoutingModule"/>
      <add name="UrlRoutingModule"
          type="System.Web.Routing.UrlRoutingModule,
              System.Web.Routing, Version=0.0.0.0,
              Culture=neutral, PublicKeyToken=31BF3856AD364E35"
      />
    </modules>

    <!-- ... -->

  </system.webServer>
</configuration>

```

The `UrlRoutingModule` passes the URL to the `RouteCollection` class, which inherits `RouteBase`. A method named `GetRouteData` will parse the request URL and determine the route that is requested. Then, the `IRouteHandler` associated with the current route is called.

The main objective of this `IRouteHandler` is to return an `IHttpHandler` that will process the request. The `GetHttpHandler` method of `IRouteHandler` will receive an instance to the current `RequestContext` and will usually return a `MvcHandler` instance of the `System.Web.Mvc` namespace. This `MvcHandler` instance will then process the current HTTP request by instantiating the controller that is defined in the active route.

Route patterns

A route is built on placeholders that are mapped to values that are parsed from the URL. These values can eventually be filled in with default values that you specify when you create the route. These placeholders, or URL parameters, are defined by enclosing them in braces: `{ }` and `}`. For example, `{name}` will define a URL parameter called `name`. URLs can be delimited by the `/` character. Everything else in the route pattern is treated as a constant value when parsing the request URL.

You can also combine multiple URL parameters by separating them with a constant value. For example, `{id}-{name}.aspx` is a valid route pattern that contains two URL parameters with names `id` and `name`. Note that `{id}{name}.aspx` would not be a valid routing pattern, as ASP.NET routing cannot determine where to separate the value for the `id` variable from the value for the `name` variable. Some examples of valid route patterns and possible matching URLs are given here:

Routing pattern	Possible matching URL
{controller}/{action}/{id}	/Products/Show/All
{controller}/{action}/{id}.aspx	/Products/Show/All.aspx
archive/{year}-{month}/{title}.aspx	/archive/2008-07/BlogPost.aspx
{language}-{country}/{controller}/{action}/{id}	/en-us/Products/Show/All
{department}/{title}.aspx	/Sales/Overview.aspx

Defining routes

Routes are defined on application startup, which is typically triggered by the `Application_Start` event in the `Global.asax` file. This event is called when the web application is first started. Make sure that you register all of the routes in a separate, static method. This will enable you to write unit tests more easily. Note that the first matching route pattern will always be used. Here's an example of a `Global.asax` file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class GlobalApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "ProductShow",                // Route name
                "Product/{id}-{title}.asp",    // URL with parameters
                new {                           // Parameter defaults
                    controller = "Product",
                    action = "Show",
                    id = "",
                    title = ""
                },
                new { id = @"[\d.*]" }          // Parameter constraints
            );

            routes.MapRoute(
```

```

        "Default", // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { // Parameter defaults
            controller = "Home",
            action = "Index",
            id = ""
        }
    );
}

protected void Application_Start()
{
    RegisterRoutes(RouteTable.Routes);
}
}

```

Routes are registered in the application's `Application_Start` event handler. Routes are added to the `RouteCollection` by calling the `MapRoute` method. The `MapRoute` method accepts two, three, or four arguments. The first argument is the route name, for example, `ProductShow`. The second argument is the route pattern, for example, `Product/{id}-{title}.aspx`. As an optional third argument, you can pass some default URL parameters. For example, if no action URL parameter is found, this route will default to the `Show` action. The fourth, and the optional, argument is used to force constraints on the URL parameters. For example, `new { id = @"[\d.*]" }` will force the `id` parameter to be numeric. If the parameter does not match the regular expression, the route will not be used; instead, the next route in the `RouteCollection` will be evaluated.

Note that we also used the `IgnoreRoute` method of `RouteCollection`. This method disables ASP.NET routing for a specific route pattern. In this case, any HTTP handler with the extension `.axd` is ignored by ASP.NET routing.

Let's see how ASP.NET routing would handle the following example route definition:

```

routes.MapRoute(
    "ProductShow", // Route name
    "Product/{id}-{title}.aspx", // URL with parameters
    new { // Parameter defaults
        controller = "Product",
        action = "Show",
        id = "",
        title = ""
    },
    new { id = @"[\d.*]" } // Parameter constraints
);

```

Request URL	Parameter values
/Product/12-RubberDuck.aspx	controller = Product action = Show id = 12 title = RubberDuck
/ Product/aa-RubberDuck.aspx	No match because of parameter constraint: the string aa does not match [\d.*]
/ Product/RubberDuck.aspx	No match

Parameter constraints

In the previous example, a URL parameter constraint was set on the `id` parameter as a regular expression string. This constraint is evaluated by the routing engine. Another option in passing URL constraints is implementing the class interface, `IRouteConstraint`. The `IRouteConstraint` interface contains a method named `Match`, which returns a boolean value that indicates whether the value is valid or not.

The following class will allow a route to be matched only if the user is authenticated:

```
public class AuthenticatedRouteConstraint : IRouteConstraint
{
    #region IRouteConstraint Members
    public bool Match(HttpContextBase httpContext, Route route,
        string parameterName, RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        // Match only when user is authenticated
        return httpContext.Request.IsAuthenticated;
    }
    #endregion
}
```

We can now add a `Secret` route to the route table by using the `AuthenticatedRouteConstraint` class as a constraint:

```
routes.MapRoute(
    "Secret",
    "Secret/{action}",
    new { controller = "Secret", action = "Index" },
    new { authenticated = new AuthenticatedRouteConstraint() }
);

routes.MapRoute(
```

```

        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = "" },
        new { controller = "^.*(?!Secret)$" }
    );

```

Note that the default route has been modified with a constraint too:

```
new { controller = "^.*(?!Secret)$" }
```

The `AuthenticatedRouteConstraint` class will not match the first route for an anonymous user, but a later route might map an anonymous user to the same controller and action method. The default route will actually do this, as it does not know anything about authentication. To make the default route ignore the `Secret` controller, a regular expression constraint has been added. You can find more information on regular expressions at www.regular-expressions.info

Catch-all routes

A situation may occur where there's a variable number of URL segments, separated by the `/` character. You can mark the last URL parameter in a routing pattern as a catch-all parameter. For example, the routing pattern `/Blog/Show/{*posttitle}` will make the URL parameter named `posttitle` the catch-all parameter. If a URL is requested, for example `/Blog/Show/More-about-catch-all-routes`, the catch-all parameter value will be populated with `More-about-catch-all-routes`. In a regular route, ASP.NET routing would try to split this string at each character, whereas, in a catch-all route, this string is passed as a whole to the controller's action method. The above route can be defined as follows:

```

routes.MapRoute(
    "BlogPost",
    "Blog/Show/{*posttitle}",
    new { controller = "Home", action = "Index", id = "" }
);

```

An example of a catch-all route can be found in the sample code for this book (CatchAllExample). This sample project contains a catch-all route that is handled by the `Index` action method of `HomeController`.

Routing namespaces

In large ASP.NET MVC web applications, using multiple namespaces for controllers can be useful. For example, the general site contents can be served by controllers in the `MySite.Controllers` namespace, while products can be served by controllers in the `MySite.Controllers.Products` namespace.

Luckily, the ASP.NET MVC routing engine is flexible enough to provide routing features across namespaces. By using the `DefaultNamespaces` property of the `ControllerBuilder`, the default namespaces can be specified. This allows you to have controller classes in multiple namespaces.

```
void Application_Start(object sender, EventArgs e) {
    ControllerBuilder.Current.DefaultNamespaces.Add("MySite.
Controllers");
    ControllerBuilder.Current.DefaultNamespaces.Add("MySite.
Controllers.
    Products");
    ControllerBuilder.Current.DefaultNamespaces.Add("ThirdParty.
Controllers
    .CreditcardProcessing");
    // ...
}
```

In this example, the default route is applied over multiple namespaces. Alternatively, these namespaces can also be reflected in the URLs—a URL for the first namespace might look like `www.mysite.com/Home/About`, while a URL for the latter may look like `www.mysite.com/Products/Catalog/List/Books`.

By using routing constraints, different URLs can be mapped to different controller namespaces. The following example code will add a constraint for the `ns` routing parameter:

```
routes.MapRoute(
    "BlogDefault",
    "{ns}/{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "" },
    new { @ns = "blog" },
    new string[] { "RoutingNamespacesExample.Controllers.Blog" }
);

routes.MapRoute(
    "ProductsDefault",
    "{ns}/{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "" },
    new { @ns = "products" },
```

```

        new string[] { "RoutingNamespacesExample.Controllers.Products" }
    );
    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = "" },
        new string[] { "RoutingNamespacesExample.Controllers" }
    );

```

The above example contains three routes. The first route accepts any URL starting with `Blog`, specified as a regular expression constraint on the route. This is mapped to the `RoutingNamespacesExample.Controllers.Blog` namespace. The second route matches all URLs starting with `Products`, and is mapped to the `RoutingNamespacesExample.Controllers.Products` namespace. The last route matches any other URL mapping, and is mapped to the `RoutingNamespacesExample.Controllers` namespace.

Note that this approach expects all views to be in `/Views/<controllername>/<action>.aspx`. If you want to have views per namespace and per controller, organize the views in `/Views/<namespace>/<controllername>/<action>.aspx` and make sure that every `ViewResult` is created with the full path to the view:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MySite.Controllers.Products {
    public class CatalogController : Controller
    {
        public ActionResult List(string category)
        {
            // ...

            return View(@"~/Views/Products/Catalog/List.aspx");
        }
    }
}

```

An example of routing with namespaces can be found in the sample code for this book (`RoutingNamespacesExample`). This sample project contains three different routes that map to different namespaces.

Combining ASP.NET MVC and ASP.NET in one web application

By default, routes are used to determine which handler will handle a specific request. Each route that is defined will map to a certain MVC controller. Even if you have an ASP.NET Webforms page named `ShowProducts.aspx`, the routing engine will try to map this request to the ASP.NET MVC framework.

You can prevent the routing engine from handling certain requests by defining a special route that defines the `StopRouteHandler` class as the handler. The `StopRouteHandler` object will stop all additional processing of the request as a route, and will force ASP.NET to handle the request either as an ASP.NET web page or as an ASP.NET web service. For example, the following routes will force the routing handler to process all of the `.aspx` requests for the directory `/Classic` in the regular ASP.NET way:

```
routes.Add(
    new Route("Classic/{resource}.aspx?{*requestUrl}",
        new StopRouteHandler())
);
routes.Add(
    new Route("Classic/{resource}.aspx",
        new StopRouteHandler())
);
```

Note that `HttpHandlers` can also be excluded from URL routing. A blank ASP.NET MVC application actually contains a default route that makes sure that `HttpHandlers` with the `.axd` file extension are handled by ASP.NET, and not by ASP.NET routing:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

The `IgnoreRoute` method here is short for using the `StopRouteHandler`. Actually, the route above can be rewritten as follows:

```
routes.Add(
    new Route("{resource}.axd/{*pathInfo}",
        new StopRouteHandler())
);
```

Creating URLs from routes

Routes can be used to generate URLs. Using this approach, constructing URLs is centralized in the routing engine, and changing a route will automatically change all of the hyperlinks in the web application. The `RouteCollection` object defines a `GetVirtualPath` method which looks for the first route in the route table that matches the parameters that the method was given.

Here's an example route definition:

```
routes.MapRoute(
    "ProductShow",                // Route name
    "Product/{id}-{title}.aspx",  // URL with parameters
    new {                          // Parameter defaults
        controller = "Product",
        action = "Show",
        id = "",
        title = ""
    },
    new { id = @"\d.*" }           // Parameter constraints
);
```

A URL for this route can be created by using the following code:

```
string url = RouteData.Route.GetVirtualPath(ControllerContext,
    new RouteValueDictionary(new
    {
        controller = "Product",
        action = "Show",
        id = 12,
        title = "RubberDuck"
    }
    )
).VirtualPath;
```

The `GetVirtualPath` method returns a `VirtualPathData` instance, which contains some information on the current virtual path, such as the URL and route data. The generated URL, which can be found in the `VirtualPath` property, will be `Product/12-RubberDuck.aspx`.

Summary

In this chapter, we have seen what ASP.NET routing is. We've also seen the difference between ASP.NET routing and URL rewriting, and how the `UrlRoutingModule` fits in the request life cycle.

Another thing we've learned is how routes are mapped to route patterns, and controller action methods. We've also learned how routes are defined, what a catch-all parameter is, and what parameter constraints are. We've used parameter constraints in an example that routes requests into different namespaces.

Finally, we've seen how an ASP.NET MVC application can be combined with ASP.NET Webforms application by setting up the `StopRouteHandler`.

6

Customizing and Extending the ASP.NET MVC Framework

One of the driving goals for the ASP.NET MVC framework has been to create a flexible framework in which every component can be extended or replaced by a custom solution, whether developed by you or obtained from a third-party vendor. This chapter describes how you can customize and extend the ASP.NET MVC framework: from creating a control and creating a custom `ActionResult` to creating your own view engine.

You will learn the following in this chapter:

- How to extend the ASP.NET MVC framework
- How to create a control, or a so-called partial view
- More about filter attributes and how to create one
- How to create a custom `ActionResult` that displays an image containing text based on a controller's action method
- How to create your own `ViewEngine` and `IView`, supporting simple HTML markup that contains entries from the `ViewData` dictionary

Creating a control

When building applications, you probably also build controls. Controls are re-usable components that contain functionality that can be re-used in different locations. In ASP.NET Webforms, a control is much like an ASP.NET web page. You can add existing web server controls and markup to a custom control and define properties and methods for it. When, for example, a button on the control is clicked, the page is posted back to the server that performs the actions required by the control.

The ASP.NET MVC framework does not support ViewState and postbacks, and therefore, cannot handle events that occur in the control. In ASP.NET MVC, controls are mainly re-usable portions of a view, called **partial views**, which can be used to display static HTML and generated content, based on ViewData received from a controller. In this topic, we will create a control to display employee details. We will start by creating a new ASP.NET MVC application using **File | New | Project...** in Visual Studio, and selecting **ASP.NET MVC Application** under **Visual C# - Web**. First of all, we will create a new Employee class inside the **Models** folder. The code for this Employee class is:

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Department { get; set; }
}
```

On the home page of our web application, we will list all of our employees. In order to do this, modify the Index action method of the HomeController to pass a list of employees to the view in the ViewData dictionary. Here's an example that creates a list of two employees and passes it to the view:

```
public ActionResult Index()
{
    ViewData["Title"] = "Home Page";
    ViewData["Message"] = "Our employees welcome you to our site!";
    List<Employee> employees = new List<Employee>
    {
        new Employee{
            FirstName = "Maarten",
            LastName = "Balliauw",
            Email = "maarten@maartenballiauw.be",
            Department = "Development"
        },
        new Employee{
            FirstName = "John",
            LastName = "Kimble",
        }
    };
}
```

```

        Email = "john@example.com",
        Department = "Development"
    }
};
return View(employees);
}

```

The corresponding view, `Index.aspx` in the **Views | Home** folder of our ASP.NET MVC application, should be modified to accept a `List<Employee>` as a model. To do this, edit the code behind the `Index.aspx.cs` file and modify its contents as follows:

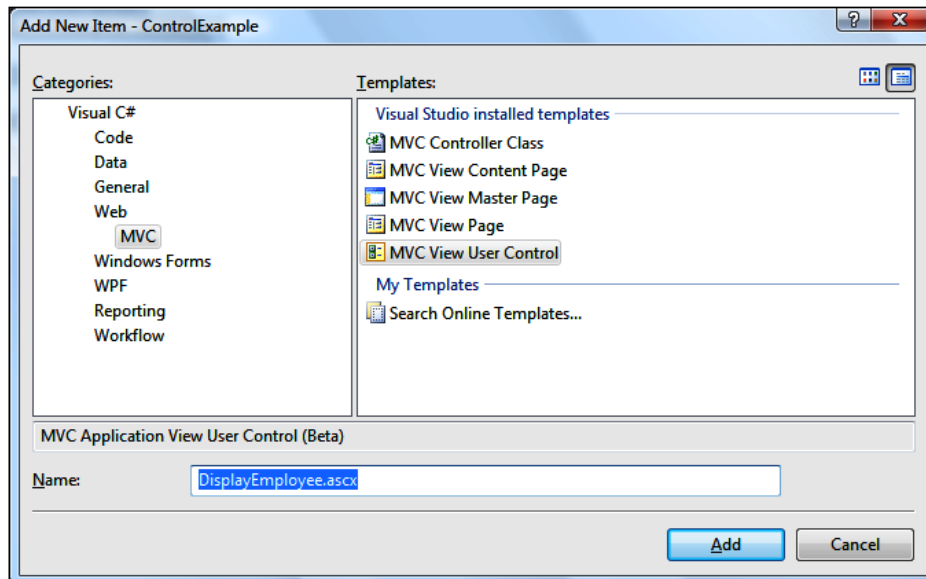
```

using System.Collections.Generic;
using System.Web.Mvc;
using ControlExample.Models;
namespace ControlExample.Views.Home
{
    public partial class Index : ViewPage<List<Employee>>
    {
    }
}

```

In the `Index.aspx` view, we can now use this list of employees. Because we will display details of more than one employee somewhere else in our ASP.NET MVC web application, let's make this a partial view.

Right-click the **Views | Shared** folder, click on **Add | New Item...** and select the **MVC View User Control** item template under **Visual C# | Web | MVC**. Name the partial view, **DisplayEmployee.ascx**.



The ASP.NET MVC framework provides the flexibility to use a strong-typed version of the `ViewUserControl` class, just as the `ViewPage` class does. The key difference between `ViewUserControl` and `ViewUserControl<T>` is that with the latter, the type of view data is explicitly passed in, whereas the non-generic version will contain only a dictionary of objects. Because the `DisplayEmployee.aspx` partial view will be used to render items of the type `Employee`, we can modify the `DisplayEmployee.ascx` code behind the file `DisplayEmployee.ascx.cs` and make it strong-typed:

```
using ControlExample.Models;
namespace ControlExample.Views.Shared
{
    public partial class DisplayEmployee :
        System.Web.Mvc.ViewUserControl<Employee>
    {
    }
}
```

In the view markup of our partial view, the model can now be easily referenced. Just as with a regular `ViewPage`, the `ViewUserControl` will have a `ViewData` property containing a `Model` property of the type `Employee`. Add the following code to `DisplayEmployee.ascx`:

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeBehind="DisplayEmployee.ascx.cs"
    Inherits="ControlExample.Views.Shared.DisplayEmployee" %>
<%=Html.Encode(Model.LastName)%>, <%=Html.Encode(Model.FirstName)%><br />
<em><%=Html.Encode(Model.Department)%></em>
```

The control can now be used on any view or control in the application. In the **Views | Home | Index.aspx** view, use the `Model` property (which is a `List<Employee>`) and render the control that we have just created for each employee:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.
Master" AutoEventWireup="true" CodeBehind="Index.aspx.cs"
Inherits="ControlExample.Views.Home.Index" %>
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2><%= Html.Encode(ViewData["Message"]) %></h2>
    <p>Here are our employees:</p>
    <ul>
        <% foreach (var employee in Model) { %>
        <li>
            <% Html.RenderPartial("DisplayEmployee", employee); %>
        </li>
        <% } %>
    </ul>
</asp:Content>
```

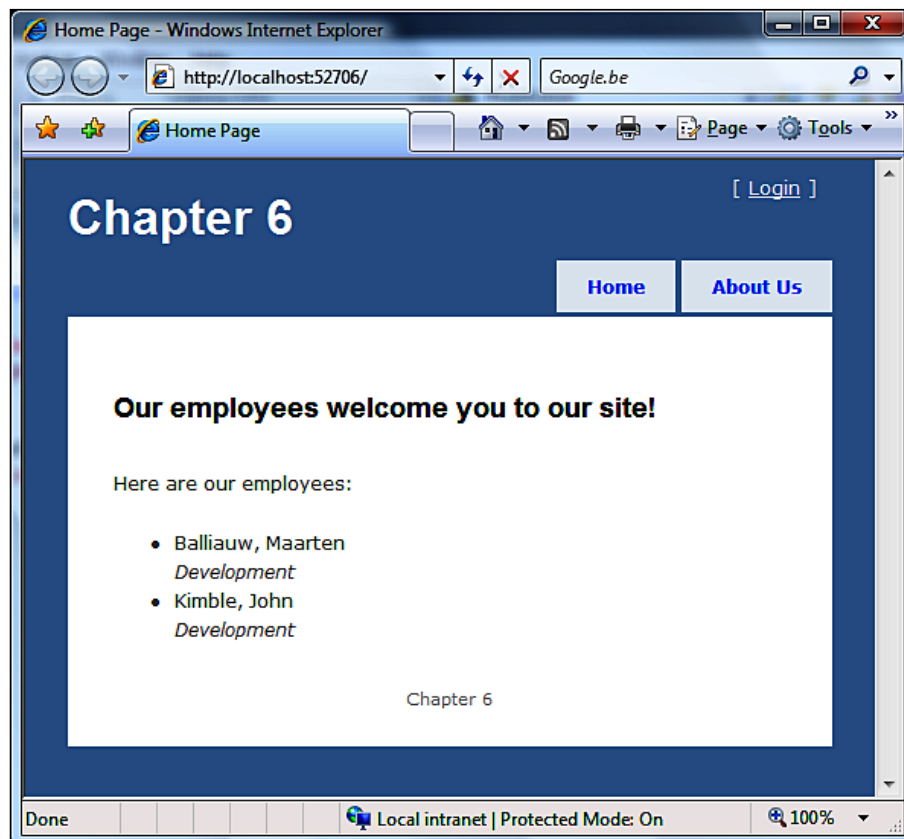
In case the control's `ViewData` type is equal to the view page's `ViewData` type, another method of rendering can also be used. This method is similar to ASP.NET Webforms controls, and allows you to specify a control as a tag. Optionally, a `ViewDataKey` can be specified. The control will then fetch its data from the `ViewData` dictionary entry having this key.

```
<uc1:EmployeeDetails ID="EmployeeDetails1"
    runat="server"
    ViewDataKey="..." />
```

For example, if the `ViewData` contains a key `emp` that is filled with an `Employee` instance, the user control could be rendered using the following markup:

```
<uc1:EmployeeDetails ID="EmployeeDetails1"
    runat="server"
    ViewDataKey="emp" />
```

After running the ASP.NET MVC web application, the result will appear as shown in the following screenshot:



Creating a filter attribute

An action filter is an attribute that can be applied to a controller class or an action method. Whenever the controller or action method is called, the action filter will be triggered both before and after the execution. Typically, action filters are used for solving problems that can occur in more than one class—the so called cross-cutting concerns. A typical cross-cutting concern is output caching or authentication—both can be required for more than one action method. More information about action filters can be found in Chapter 4, *Components in the ASP.NET MVC Framework*.

One cross-cutting concern of an action method might be logging. For example, one wants to log when an action was called, and whether its result was executed, in a log file as shown here:

```
[2008-09-02 - 03:03:13] Controller: Home; Action: Index; Action
executing...
[2008-09-02 - 03:03:13] Controller: Home; Action: Index; Action
executed.
[2008-09-02 - 03:03:13] Controller: Home; Action: Index; Result
executing...
[2008-09-02 - 03:03:15] Controller: Home; Action: Index; Result
executed.
[2008-09-02 - 03:04:42] Controller: Account; Action: Login; Action
executing...
[2008-09-02 - 03:04:42] Controller: Account; Action: Login; Action
executed.
[2008-09-02 - 03:04:42] Controller: Account; Action: Login; Result
executing...
[2008-09-02 - 03:04:43] Controller: Account; Action: Login; Result
executed.
[2008-09-02 - 03:04:44] Controller: Home; Action: About; Action
executing...
[2008-09-02 - 03:04:44] Controller: Home; Action: About; Action
executed.
[2008-09-02 - 03:04:44] Controller: Home; Action: About; Result
executing...
[2008-09-02 - 03:04:44] Controller: Home; Action: About; Result
executed.
```

To achieve this, a filter attribute can be created by implementing the `IActionFilter` and `IResultFilter` interfaces, and optionally overloading the `FilterAttribute` class. An action filter allows you to run code before and after an action method is called, but before the result of the action method is executed. A result filter is very similar to an action filter except that it is executed before and after the result is returned from the action that has been executed.

The `IActionFilter` interface defines two methods:

- `OnActionExecuting`
Runs before the action method is executed.
- `OnActionExecuted`
Runs after the action method is executed, but before the `ActionResult` is executed.

The `IResultFilter` interface defines two methods:

- `OnResultExecuting`
Runs before the action method's action result is executed
- `OnResultExecuted`
Runs after the action method's action result is executed.

Let's create a class called `LoggingAttribute` which implements these two interfaces, and will run whenever an action is executing or an action result is executing. The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`ActionFilterExample`).

First of all, let's define the class and apply the `AttributeUsage` attribute to it. This attribute tells the compiler that the `LoggingAttribute` we are creating can only be applied to classes and methods. The `LoggingAttribute` class implements `IActionFilter` and `IResultFilter`. We also add a property named `LogName`, which will hold the path to our log file.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    Inherited = true, AllowMultiple = true)]
public class LoggingAttribute : FilterAttribute, IActionFilter,
    IResultFilter
{
    #region Properties
    public string LogName { get; set; }
    #endregion
}
```

Because we will be logging some things, let's also add a method that writes a log message to the file referenced by the `LogName` property. This method will simply open the file, append a new line to it, and close it again.

```
private void LogMessage(string controller, string action, string
    message)
{
    if (!string.IsNullOrEmpty(LogName))
    {
```

```
        TextWriter writer = new StreamWriter(LogName, true);
        writer.WriteLine("[{0}] Controller: {1}; Action: {2}; {3}",
            DateTime.Now.ToString("yyyy-MM-dd - hh:mm:ss"),
            controller, action, message);
        writer.Close();
    }
}
```

Now, it is time to implement the `IActionFilter` and `IResultFilter` interfaces. For the `IActionFilter`, we'll add the `OnActionExecuting` and `OnActionExecuted` methods. For `IResultFilter`, we'll add the `OnResultExecuting` and `OnResultExecuted` methods. All of these methods will use the `LogMessage` method that we've just created and pass in some information for logging to the file.

```
public void OnActionExecuting(ActionExecutingContext filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Action executing..."
    );
}

public void OnActionExecuted(ActionExecutedContext filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Action executed."
    );
}

public void OnResultExecuting(ResultExecutingContext filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Result executing..."
    );
}

public void OnResultExecuted(ResultExecutedContext filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Result executed."
    );
}
```

The source of information that is being logged originates in the parameter passed to the On... methods. This is always an object containing information about the current execution context, such as the controller that is executing, the view that is being rendered, and HTTP request data.

Here's the full `LoggingAttribute` class that we have been creating:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    Inherited = true, AllowMultiple = true)]
public class LoggingAttribute : FilterAttribute, IActionFilter,
    IResultFilter
{
    #region Properties
    public string LogName { get; set; }
    #endregion
    #region Helper methods
    private void LogMessage(string controller, string action, string
        message)
    {
        if (!string.IsNullOrEmpty(LogName))
        {
            TextWriter writer = new StreamWriter(LogName, true);
            writer.WriteLine("[{0}] Controller: {1}; Action: {2}; {3}",
                DateTime.Now.ToString("yyyy-MM-dd - hh:mm:ss"),
                controller, action, message);
            writer.Close();
        }
    }
    #endregion
    #region IActionFilter Members
    public void OnActionExecuting(ActionExecutingContext
        filterContext)
    {
        LogMessage(
            filterContext.RouteData.Values["controller"].ToString(),
            filterContext.RouteData.Values["action"].ToString(),
            "Action executing..."
        );
    }
    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        LogMessage(
```

```
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Action executed."
    );
}
#endregion
#region IResultFilter Members
public void OnResultExecuting(ResultExecutingContext
    filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Result executing..."
    );
}
public void OnResultExecuted(ResultExecutedContext filterContext)
{
    LogMessage(
        filterContext.RouteData.Values["controller"].ToString(),
        filterContext.RouteData.Values["action"].ToString(),
        "Result executed."
    );
}
#endregion}
```

The filter can now be applied to any controller, which in turn will apply the filter to all of the action methods, or to individual action methods:

```
[Logging(LogName = "C:\\temp\\ApplicationLog.log")]
public class HomeController : Controller
{
    // ...

    [Logging(LogName = "C:\\temp\\ActionLog.log")]
    public ActionResult SomeAction() {
        // ...
    }
}
```

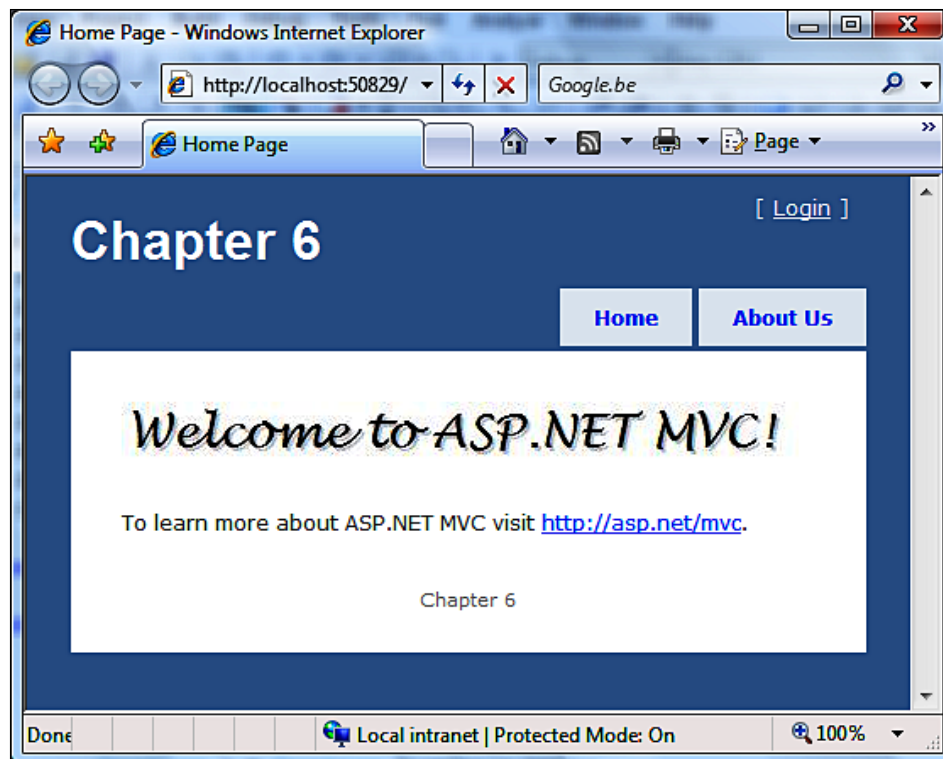
Now, when the `SomeAction` action method of the `HomeController` is called, a log entry will be created when the view is being rendered.

Creating a custom ActionResult

The ASP.NET MVC framework's action method implements the concept of returning an `ActionResult` instance, which will typically render a specific view, or redirect the user to a different location on the web site. An `ActionResult` that renders a view is returned as a `RenderViewResult`. The `ExecuteResult()` method is called in order to render specific contents to the HTTP response stream.

An `ActionResult` can take any form, as we have seen in Chapter 4, as long as it has something to do with the HTTP response stream. For example, you can create a `FileDownloadResult` that streams a file on the HTTP response stream, or a `PermanentRedirectResult` that renders HTTP status code 302.

One of the problems that many web designers face is the fact that a user's web browser may or may not have the specific fonts needed to display the contents. This may be a problem, say, if the web designer wants to style a title element in some exotic font face. Because the ASP.NET MVC framework has a modular architecture, a custom `ActionResult` class can easily be created to achieve this goal. This custom `ActionResult` will render a JPEG image based on input text, which can be used to display a page title. This `ActionResult` class will be named `ImageResult`.



The example in this topic is based on an ASP.NET MVC web application that can be found in the sample code for this book (CustomActionResultExample). The **Code** folder contains the `ImageResult` that we will build. The new `ImageResult` class will inherit the abstract class `ActionResult` and implement its `ExecuteResult` method. This method basically performs communication over the HTTP response stream. It accepts an `Image` object as a property as well as an `ImageFormat`. This means that a custom image can easily be rendered to the HTTP response stream, whether as JPEG, BMP, PNG, or GIF.

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Web.Mvc;

namespace CustomActionResultExample.Code
{
    public class ImageResult : ActionResult
    {
        public ImageResult() { }

        public Image Image { get; set; }
        public ImageFormat ImageFormat { get; set; }

        public override void ExecuteResult(ControllerContext context)
        {
            // verify properties
            if (Image == null)
            {
                throw new ArgumentNullException("Image");
            }
            if (ImageFormat == null)
            {
                throw new ArgumentNullException("ImageFormat");
            }

            // output
            context.HttpContext.Response.Clear();

            if (ImageFormat.Equals(ImageFormat.Bmp)) context.
HttpContext.Response.ContentType = "image/bmp";
            if (ImageFormat.Equals(ImageFormat.Gif)) context.
HttpContext.Response.ContentType = "image/gif";
            if (ImageFormat.Equals(ImageFormat.Icon)) context.
HttpContext.Response.ContentType = "image/vnd.microsoft.icon";
            if (ImageFormat.Equals(ImageFormat.Jpeg)) context.
HttpContext.Response.ContentType = "image/jpeg";
            if (ImageFormat.Equals(ImageFormat.Png)) context.
HttpContext.Response.ContentType = "image/png";
```

```

        if (ImageFormat.Equals(ImageFormat.Tiff)) context.
HttpContext.Response.ContentType = "image/tiff";
        if (ImageFormat.Equals(ImageFormat.Wmf)) context.
HttpContext.Response.ContentType = "image/wmf";
        Image.Save(context.HttpContext.Response.OutputStream,
                    ImageFormat);
    }
}

```

The `ImageResult` class defines two properties: `Image` and `ImageFormat`. These properties can be set in the `ImageResult` constructor. When the `ImageResult` is executed in the `ExecuteResult` method, the in-memory image is rendered to the HTTP response stream in the specified image format.

Rendering the page title image will be done by a `PageTitleController` which has one action method, `ShowTitle`, that builds a bitmap image and returns it as an `ImageResult`.

```

using System.Drawing;
using System.Drawing.Imaging;
using System.Web.Mvc;
using CustomActionResultExample.Code;
namespace CustomActionResultExample.Controllers
{
    public class PageTitleController : Controller
    {
        public ActionResult ShowTitle(string pageTitle, int width,
                                     int height)
        {
            // Create bitmap
            Bitmap bmp = new Bitmap(width, height);
            Graphics g = Graphics.FromImage(bmp);
            g.FillRectangle(Brushes.White, 0, 0, width, height);
            // Render light gray background
            g.DrawString(pageTitle, new Font("Lucida Handwriting",
                                             height / 2),
                        Brushes.LightGray, new PointF(2, 2));
            // Render black text on top
            g.DrawString(pageTitle, new Font("Lucida Handwriting",
                                             height / 2),
                        Brushes.Black, new PointF(0, 0));
            // Return ImageResult
            return new ImageResult { Image = bmp, ImageFormat =
                                   ImageFormat.Jpeg };
        }
    }
}

```

The `ShowTitle` action method creates a new in-memory bitmap and renders a text and text shadow on a white background. This bitmap is then passed into a new `ImageResult` instance, which will render the image to the HTTP response stream.

As an extra feature, this image-rendered page title can be added to the web page in an easy manner.

```
<%=Html.PageTitle("Welcome to ASP.NET MVC!", 400, 40)%>
```

This `HtmlHelper` extension method will render an HTML image tag such as ``. It is implemented as follows:

```
using System.Web.Mvc;
using CustomActionResultExample.Controllers;
using Microsoft.Web.Mvc;

namespace CustomActionResultExample.Code
{
    public static class PageTitleHelper
    {
        public static string PageTitle(this HtmlHelper helper, string
            pageTitle, int width, int height)
        {
            string url = LinkBuilder.BuildUrlFromExpression
                <PageTitleController>(helper.ViewContext.
                    RequestContext, helper.RouteCollection,
                    c => c.ShowTitle(pageTitle, width, height));

            return string.Format("<img src=\"{0}\" width=\"{1}\"
                height=\"{2}\" alt=\"{3}\" />", url, width,
                height, pageTitle);
        }
    }
}
```



We have used a new namespace here (`Microsoft.Web.Mvc`) to make use of the `LinkBuilder` class. This namespace contains several classes that may be included in the ASP.NET MVC framework in future, but are currently not considered to be stable by the ASP.NET MVC development team. The MVC features can be downloaded from the official CodePlex site at <http://www.codeplex.com/aspnet>.

If all of the classes are in place, we can now add an image-based title in our view in an easy, intuitive manner. The following code will replace the default home page by an enhanced version using our newly-created `ImageResult`. The `HtmlHelper` class now has a new method, `PageTitle`, which we can use to create a title image of a specified width and height.

```
<asp:Content ID="indexContent"
    ContentPlaceHolderID="MainContent"
    runat="server">
    <%=Html.PageTitle(ViewData["Message"].ToString(), 400, 40)%>
    <p>
        To learn more about ASP.NET MVC visit <a
        href="http://asp.net/mvc" title="ASP.NET MVC
        Website">http://asp.net/mvc</a>.
    </p>
</asp:Content>
```

After running the application, the index page will look like the screenshot presented earlier in this topic.

Creating a ViewEngine

A `ViewEngine` maps view names to actual files on the web server and instantiates a `View` if one is found. By default, views are located in the **Views | ControllerName** project folder, or in the **Views | Shared** folder. There are some custom `ViewEngine` implementations available on the Internet (NHaml, Spark, and so on; you will find links to these in Appendix C of this book); we will be building a custom `ViewEngine` and `View` implementation.

All `ViewEngine` implementations for the ASP.NET MVC framework implement the `IViewEngine` interface:

```
public interface IViewEngine
{
    ViewEngineResult FindPartialView(ControllerContext
        controllerContext, string partialViewName);
    ViewEngineResult FindView(ControllerContext controllerContext,
        string viewName, string masterName);
}
```

The only responsibility an `IViewEngine` implementation has is to find a view or a partial view in the application. If a view has not been found, the implementation should return a list of searched locations. If a view has been found, a `ViewEngineResult` is returned.

When a view is required to be rendered, each registered `IViewEngine` is consulted (in the order in which they were registered) until the ASP.NET MVC framework finds one that returns a view that can be rendered.

The `WebFormsViewEngine` (ASP.NET MVC's default) searches the following virtual paths for views or partial views:

- `~/Views/<controllerName>/<viewName>.aspx`
- `~/Views/<controllerName>/<viewName>.ascx`
- `~/Views/Shared/<viewName>.aspx`
- `~/Views/Shared/<viewName>.ascx`

Master pages are searched for in the following virtual paths:

- `~/Views/<controllerName>/<masterName>.master`
- `~/Views/Shared/<masterName>.master`

In order to create a custom `IViewEngine`, the tone can overload the base class, `VirtualPathProviderViewEngine`, instead of implementing the `IViewEngine` interface. The `VirtualPathProviderViewEngine` class provides the base functionality for searching a view on a file system.

The example in this topic is based on an ASP.NET MVC web application which can be found in the sample code for this book (`CustomViewEngine`).

Let's start creating a `SimpleViewEngine` by overloading the `VirtualPathProviderViewEngine`. In the constructor, set the paths where the master and view pages can be found. The `SimpleViewEngine` will search for views and partial views in the same locations that the `WebFormsViewEngine` does, except that it searches for `.htm` or `.html` files. Master page support is not available; hence the empty path is passed in the constructor.

Next, override the two methods: `CreatePartialView()` and `CreateView()`. These methods are used to instantiate a view based on the path defined in this `SimpleViewEngine`. `CreatePartialView()` and `CreateView()` return a `SimpleView`, which is our own view implementation, on which we'll focus right away.

```
using System.Web.Mvc;

namespace CustomViewEngine.Core
{
    public class SimpleViewEngine : VirtualPathProviderViewEngine
    {
        #region Constructor
```

```

public SimpleViewEngine() : base()
{
    base.MasterLocationFormats = new string[] { "" };
    base.ViewLocationFormats = new string[] {
        "~/Views/{1}/{0}.htm",

        "~/Views/{1}/{0}.html",

        "~/Views/Shared/{0}.htm",

        "~/Views/Shared/{0}.html"
    };
    base.PartialViewLocationFormats = ViewLocationFormats;
}
#endregion
#region VirtualPathProviderViewEngine Members
protected override IView CreatePartialView(ControllerContext
    controllerContext, string partialPath)
{
    return new SimpleView(partialPath);
}
protected override IView CreateView(ControllerContext
    controllerContext, string viewPath, string masterPath)
{
    return new SimpleView(viewPath);
}
#endregion
}
}

```

IView is the interface that is used for defining a view. A view is responsible for rendering itself to a `TextWriter` instance, which will probably be the HTTP response stream. Let's create our `SimpleView` class. First of all, implement the `IView` interface. This interface defines the `Render()` method on which we'll focus later. Also, add a constructor that can be used by the `SimpleViewEngine` that we created earlier, and a `ViewPath` property, for the sake of convenience when debugging.

```

using System;
using System.Collections;
using System.IO;
using System.Reflection;
using System.Text.RegularExpressions;
using System.Web.Mvc;

```

```
namespace CustomViewEngine.Core
{
    public class SimpleView : IView
    {
        #region Private fields
        private string viewPath;
        #endregion

        #region Constructor
        public SimpleView(string viewPath)
        {
            this.viewPath = viewPath;
        }
        #endregion

        #region Public properties
        public string ViewPath {
            get { return this.viewPath; }
        }
        #endregion

        #region IView Members
        public virtual void Render(ViewContext viewContext,
            TextWriter writer)
        {
            // ...
        }
        #endregion
    }
}
```

Next, let's implement the `Render()` method. This is passed the `ViewContext` and `TextWriter` instances. The first instance contains the `ViewData` that we are receiving from the controller, along with some other properties. The latter will probably be the HTTP response stream. In our implementation of `Render()`, the view source code is evaluated using a regular expression, which will look for things such as `{ $ViewData.Message }`, and which we will map to `ViewData["Message"]` later on.

```
public virtual void Render(ViewContext viewContext, TextWriter
    writer)
{
    string viewTemplate = File.ReadAllText(
        viewContext.HttpContext.Request.MapPath(this.viewPath)
    );
}
```

```

        Regex templatePattern = new Regex(@"({\$w+((\.|[\]\w+\\]?)*)})",
            RegexOptions.Multiline);
        MatchEvaluator replaceCallback = new MatchEvaluator(m =>
            SimpleView.Resolve(m.Value,
                viewContext.ViewData).
                ToString());
        viewTemplate = templatePattern.Replace(viewTemplate,
            replaceCallback);
        writer.Write(viewTemplate);
    }

```

Note that the `MatchEvaluator` will call a method named `Resolve()` for each match that is found in the view markup. We will not go deeper into the `Resolve()` method, but it basically replaces things such as `{ ViewData.Message }` with more meaningful data found in `ViewData["Message"]`. The `Resolve()` method can be found in the following complete code for `SimpleView`:

```

using System;
using System.Collections;
using System.IO;
using System.Reflection;
using System.Text.RegularExpressions;
using System.Web.Mvc;

namespace CustomViewEngine.Core
{
    public class SimpleView : IView
    {
        #region Private fields
        private string viewPath;
        #endregion

        #region Constructor
        public SimpleView(string viewPath)
        {
            this.viewPath = viewPath;
        }
        #endregion

        #region Public properties
        public string ViewPath {
            get { return this.viewPath; }
        }
        #endregion

        #region IView Members

```

```
public virtual void Render(ViewContext viewContext,
    TextWriter writer)
{
    string viewTemplate = File.ReadAllText(viewContext.
        HttpContext.Request.MapPath
        (this.viewPath));

    Regex templatePattern = new Regex(@"({\$w+((\.|[\]\w+\\]?)*)}) "
        , RegexOptions.Multiline);
    MatchEvaluator replaceCallback = new MatchEvaluator(m =>
        SimpleView.Resolve(m.Value,
            viewContext.ViewData).
            ToString());

    viewTemplate = templatePattern.Replace
        (viewTemplate, replaceCallback);

    writer.Write(viewTemplate);
}

#endregion

#region Helper methods

public static object Resolve(string sourceString,
    object sourceObject)
{
    // Setup regular expressions engine
    Regex templatePattern = new Regex(@"({\$w+((\.|[\]\w+\\]?)*)}) ",
        RegexOptions.Multiline);

    object resolvedObject = null;
    MatchEvaluator replaceCallback = new MatchEvaluator(
        delegate(Match m)
        {
            // Split expression
            string[] expressions = m.Value.Replace("{", "")
                .Replace("$", "")
                .Replace("}", "")
                .Replace("[", ".")
                .Replace("]", ".")
                .Split('.');

            // Loop expressions
            object lastObject = sourceObject;
            string expression = "";
            for (int i = 1; i < expressions.Length; i++)
            {
                expression = expressions[i];
                if (lastObject != null)
                {
```

```
        if (lastObject is IDictionary<string,
            object>)
        {
            lastObject = ((IDictionary<string,
                object>)lastObject)[expression];
        }
        else if (lastObject is IDictionary)
        {
            lastObject = ((IDictionary)lastObject)
                [expression];
        }
        else if (lastObject is Array)
        {
            lastObject = ((Array)lastObject).GetValue
                (int.Parse(expression));
        }
        else
        {
            try
            {
                lastObject = lastObject.GetType().
                    InvokeMember(expression,
                        BindingFlags.Instance |
                        BindingFlags.Public |
                        BindingFlags.GetField |
                        BindingFlags.GetProperty,
                        null, lastObject, null);
            }
            catch (MissingMethodException)
            {
                lastObject = string.Format("Undefined:
                    {0}", m.Value);
            }
        }
    }
}

if (lastObject != null)
{
    resolvedObject = lastObject;
}
else
{
    resolvedObject = string.Format("Undefined: {0}",
        sourceString);
}
```

```
        return resolvedObject.ToString();
    }
};
// Fire up replacement engine!
templatePattern.Replace(sourceString, replaceCallback);
return resolvedObject;
}
#endregion
}
```

The `Render()` method is responsible for rendering the view. It loads the template from the file system, replaces some variables with data from the `ViewData` dictionary using a regular expression and regular expression callback, and renders the result to the provided `TextWriter`.

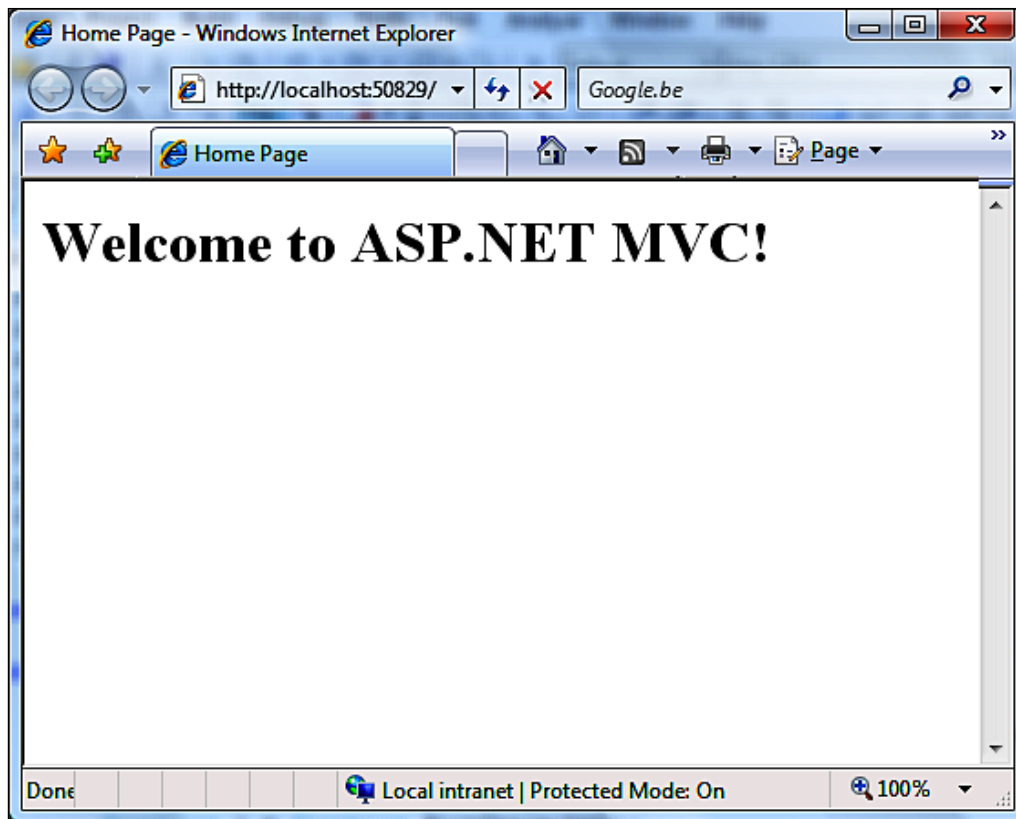
The view markup for this `SimpleViewEngine` can be found in `/Views/<controller>/<action>.htm`. The following code snippet is the view markup for **Views | Home | Index.htm**, used by the `Index` action method of the `HomeController` class.

```
<html>
  <head>
    <title>{$ViewData.Title}</title>
  </head>
  <body>
    <h1>{$ViewData.Message}</h1>
  </body>
</html>
```

For the `SimpleViewEngine` to be consulted by the ASP.NET framework, it has to be registered. This has to be done once, preferably in the `Application_Start` event handler, which can be found in the `Global.asax.cs` file, and is called only the first time the application is started:

```
protected void Application_Start()
{
    ViewEngines.Engines.Add(new SimpleViewEngine());
    RegisterRoutes(RouteTable.Routes);
}
```

Once we remove the `Index.aspx` view created by the ASP.NET MVC Visual Studio Project Template, the `Index.htm` view we created earlier will be used by the `Index` action method of the `HomeController` class. On running the application, you will see the following screen:



Summary

In this chapter, we learned how to extend the ASP.NET MVC framework. We created a control, which is also called a partial view. We also learned more about filter attributes, and also created one of our own.

We looked at how to create a custom `ActionResult`, which displays an image containing text based on a controller's action method.

Finally, we created our own `ViewEngine` and `IView`, which provided support for simple HTML markup containing entries from the `ViewData` dictionary.

7

Using Existing ASP.NET Features

Ever since Microsoft started working on the ASP.NET MVC framework, one of the primary concerns was the framework's ability to re-use as many features as possible from ASP.NET Webforms. Because ASP.NET MVC is built on top of ASP.NET, you can easily use features such as .ASPX, .ASCX, and master pages, server controls, templates, data binding, and so on. Actually, any feature except for those features requiring ViewState or postbacks from ASP.NET Webforms, can be used.

You will learn the following in this chapter:

- Using session state in the ASP.NET MVC framework
- What TempData is, and what its use is in an ASP.NET MVC web application
- What membership, authentication, and authorization are
- How to configure web site security
- How to use existing ASP.NET membership, authentication, and authorization providers
- Protecting access to specific controller action methods
- How you can make use of output caching in an ASP.NET MVC web application
- What internationalization is, and how to use it
- Mixing ASP.NET Webforms and ASP.NET MVC in one application and sharing data between both technologies
- How to add a post-build action to a project file to make sure that all views can be compiled

Session State

When working with ASP.NET, you may have used session state to store user-specific information, which is maintained between server round trips. Session state is actually a set of key-value pairs that is persisted on the server and scoped to a browser session. If different users are using your application, each user will have their own set of key-value pairs. Whenever a session times out, a user is assigned an empty session state.

Each client is identified by a unique key, which is stored in a cookie or in a request variable. On the server, this identifier is mapped to a session, which is stored in the web server's process, a session server, SQL server, or any custom session store that inherits from `System.Web.SessionState.SessionStateStoreProviderBase`.

Reading and writing session data

Session state is stored in the context information of an ASP.NET MVC application. When writing code for a controller action, you can easily access the `HttpSessionStateBase` object in the controller's `Session` property. This is accessed in the same way as which you would access any dictionary. Keys are always of type string; values are of type object. This means that you can add any type of object to the session state, as long as it is marked `[Serializable]`.



Serialization is the process of converting an object into a sequence of bits that can be stored in a file, a memory buffer, a database, or that can be transmitted across a network. The `[Serializable]` attribute marks a specific object as being serializable.

In this example, serialization is needed by the session state provider, as it may temporarily store the session data in various locations—in the memory, in a database, on a session state server, and so on. Making the class serializable allows the session state provider to store the object anywhere.

The following code uses the Session dictionary's key Message to read and write a message to session state:

```
using System.Web.Mvc;
namespace SessionStateExample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            Session["Message"] = "Hello world!";
            return View();
        }

        public ActionResult ReadData()
        {
            string message = "";
            if (Session["Message"] != null) {
                message = (string)Session["Message"];
            }
            return View();
        }
    }
}
```

Configuring session state

Every aspect of session state can be configured in your web application's web.config. For example, one can completely disable the session state:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <sessionState mode="off"/>
  </system.web>
</configuration>
```

This `sessionstate` element can contain numerous attributes; some of them have been enumerated here:

Attribute	Default	Possible Values
timeout	mode="InProc"	InProc: Enables session state storage in the Web server's memory; this is the default option and offers the best performance of all possible options. Web applications running on one web server and not requiring session state to be persisted between application restarts should use this option. StateServer: Stores session state in a Windows service called "ASP.NET State Service". This service should be enabled in the server's control panel. Using this state server, session state is hosted in an external process and is available for other web servers. This is not a redundant session store as session data can be lost when the state server is restarted. SQLServer: Session state is stored in an SQL server database and preserved between application restarts and database restarts. This is not the most performant session store, but it offers good redundancy. Custom: Enables you to use a custom storage provider. A custom session state provider inherits from the <code>System.Web.SessionState.SessionStateStoreProviderBase</code> class. Off: Completely disables session state.
timeout	timeout="20"	Will expire the session state after the configured amount of minutes.
cookieName	cookieName="ASP.NET_SessionId"	Holds the name of the cookie that will be used to store the session identifier.

Attribute	Default	Possible Values
cookieless	cookieless="UseCookies"	Specifies how cookies are used for storing the session identifier. AutoDetect: ASP.NET will try to detect if the client's browser supports cookies and falls back to using the URI if the browser does not support cookies. UseCookies: Requires using cookies. UseDeviceProfile: Uses a pre-configured profile to enable or disable cookies UseUri: Stores the session identifier in the request URI
regenerateExpiredSessionId	regenerateExpiredSessionId="True"	Enables or disables automatic generation of a new session identifier whenever a session expires. Possible values are True or False

Refer to the MSDN page at <http://msdn.microsoft.com/en-us/library/h6bb9cz9.aspx> for more information on configuring session state.

TempData

The ASP.NET MVC framework has built one extra feature on top of regular sessions: TempData. TempData is a special storage mechanism that can be used to store data across two requests. Note that the data is stored only for a single web request! This means that, unlike a standard session, the TempData entry will be removed once a second web request has been made.

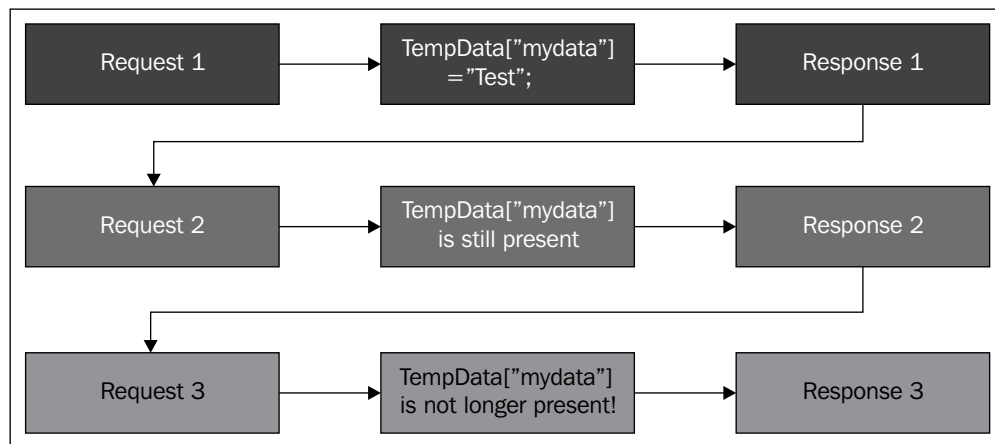
TempData is very useful for keeping values that might be needed in a second request. An example use for TempData could be pagination. Imagine that you have a search page displaying some results, and you only want to show 'x' number of items at a time. TempData can be used to save the search command while the user browses all pages. Another use could be passing data between controllers. If you are redirecting from one controller to another, and you need to pass data between them, TempData can be used to pass this data.

Using TempData in the ASP.NET MVC framework is pretty straightforward:

```
// Write:
TempData["mydata"] = "This is my data.";

// Read:
var myData = TempData["mydata"];
```

The following image illustrates the life cycle of a TempData entry:



Membership, authentication, and authorization

In the early days of the Internet, most web sites were public spaces, having information available for all users. Nowadays, most web sites offer an authentication mechanism that allows users to store private information and use members-only site features.

Authentication is the mechanism whereby a user is securely identified by a system. It provides the answer to a simple question: who is this user? Authorization is another mechanism, which is tightly coupled to authentication. Authorization is about determining the level of access for a particular user—can this user access this page? Can this user add a new client?

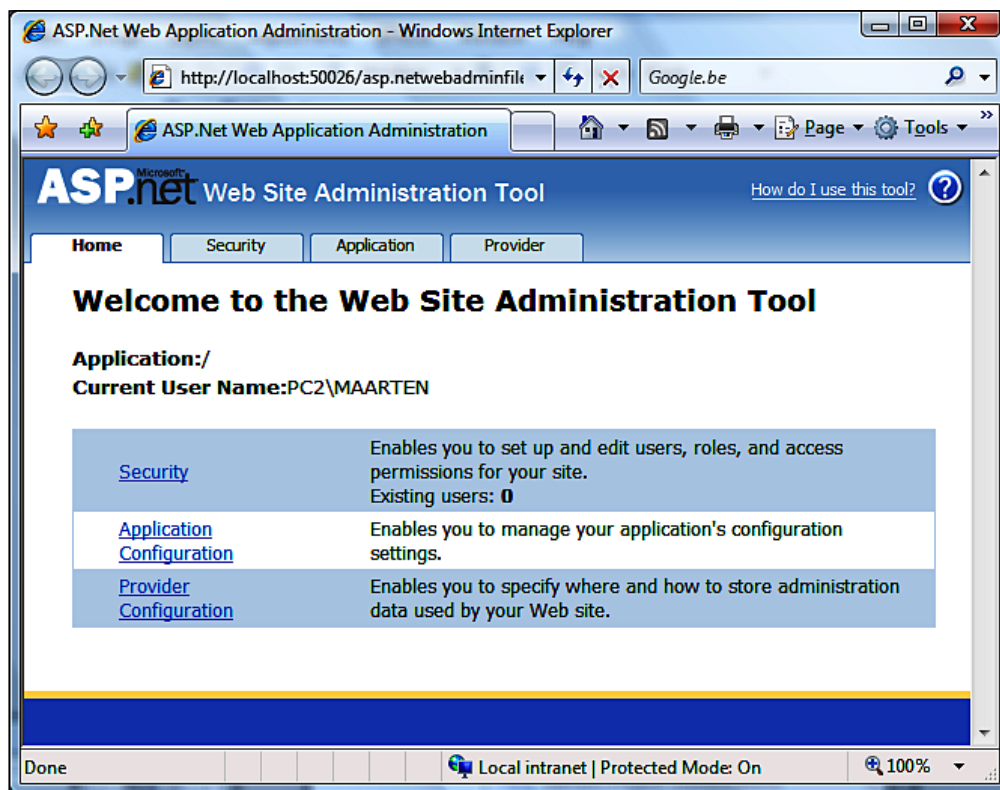
The ASP.NET framework provides several features for handling authentication and authorization. In addition, it provides some tools that make user management easier in a web application. These include wizards for configuring user management and membership, and role classes, which you can use to identify a user and grant or deny access to specific resources.

When creating a new ASP.NET MVC application, you will notice a default `AccountController` and associated views that expose classic ASP.NET membership in the ASP.NET MVC framework.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`AuthenticationExample`).

Configuring web site security

After compiling a newly-created ASP.NET MVC web application for the first time, it is possible to make use of the **Web Site Administration Tool** in order to configure your web application's security. This tool provides a web-based frontend to the web application's `web.config` file. The **Web Site Administration Tool** can be found in Visual Studio's **Project** menu under **ASP.NET Configuration**. Note that it takes a while to load this tool the first time. This is because a default membership database is created in your web application's `App_Data` directory.



To configure users and roles, use the Security configuration wizard:

1. On the **Web Site Administration Tool** home page, click on the **Security** link.
2. Click on the **Use the security Setup Wizard to configure security step by step** link.
3. Click on **Next** to continue.
4. Pick one of the following options:
 - **From the internet** – use a SQL server based user and role configuration
 - **From a local area network** – use Windows authentication based on local computer users and Active Directory
5. Complete the wizard and add some roles and users.

After completion, the **Web Site Administration Tool** can be closed, and the web application's `web.config` will automatically be updated to reflect the new configuration.

Implementing user and role based security in a controller

After the security for an ASP.NET MVC application has been configured, a controller or an action method can be protected by specifying users and/or roles that can access the controller or action method. To do this, an `IAuthorizationAttribute` can be used. An out-of-the-box implementation of this `IAuthorizationAttribute` is the `AuthorizeAttribute`.

The `AuthorizeAttribute` provides two parameters, which can both be applied at the same time:

Parameter	Description
No specified parameters	User is required to authenticate, but is allowed access regardless of his username and role
Users	A comma-separated list of usernames that are granted access; this can also be an Active Directory username in the form of <code>DOMAIN\Username</code>
Roles	A comma-separated list of roles that are granted access; this can also be an Active Directory group in the form of <code>DOMAIN\Group</code>

The following example requires a user to be authenticated before he or she can call any action method on the HomeController. This is done by adding the `[Authorize]` attribute to the Index action method. Additionally, the About action method can be called only if the authenticated user has the role **Administrator**. This is done by specifying the `[Authorize(Roles="Administrator")]` attribute.

```
using System.Web.Mvc;
namespace AuthenticationExample.Controllers
{
    [HandleError]
    [Authorize]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Title"] = "Home Page";
            ViewData["Message"] = "Welcome to ASP.NET MVC!";
            return View();
        }

        [Authorize(Roles="Administrator")]
        public ActionResult About()
        {
            ViewData["Title"] = "About Page";
            return View();
        }
    }
}
```

When the above example is run, the user will immediately be redirected to the Login action method of the AccountController class. The web application's `web.config` has been preconfigured for this. The `<forms>` element's `loginUrl` attribute has been configured to redirect unauthenticated users to the `/Account/Login` URL when authentication is required. ASP.NET will automatically add a `returnUrl` parameter to this URL to redirect a user back to the page that he or she originally requested. The AccountController class makes use of ASP.NET's `FormsAuthentication` and `MembershipProvider` classes to handle authentication and authorization.

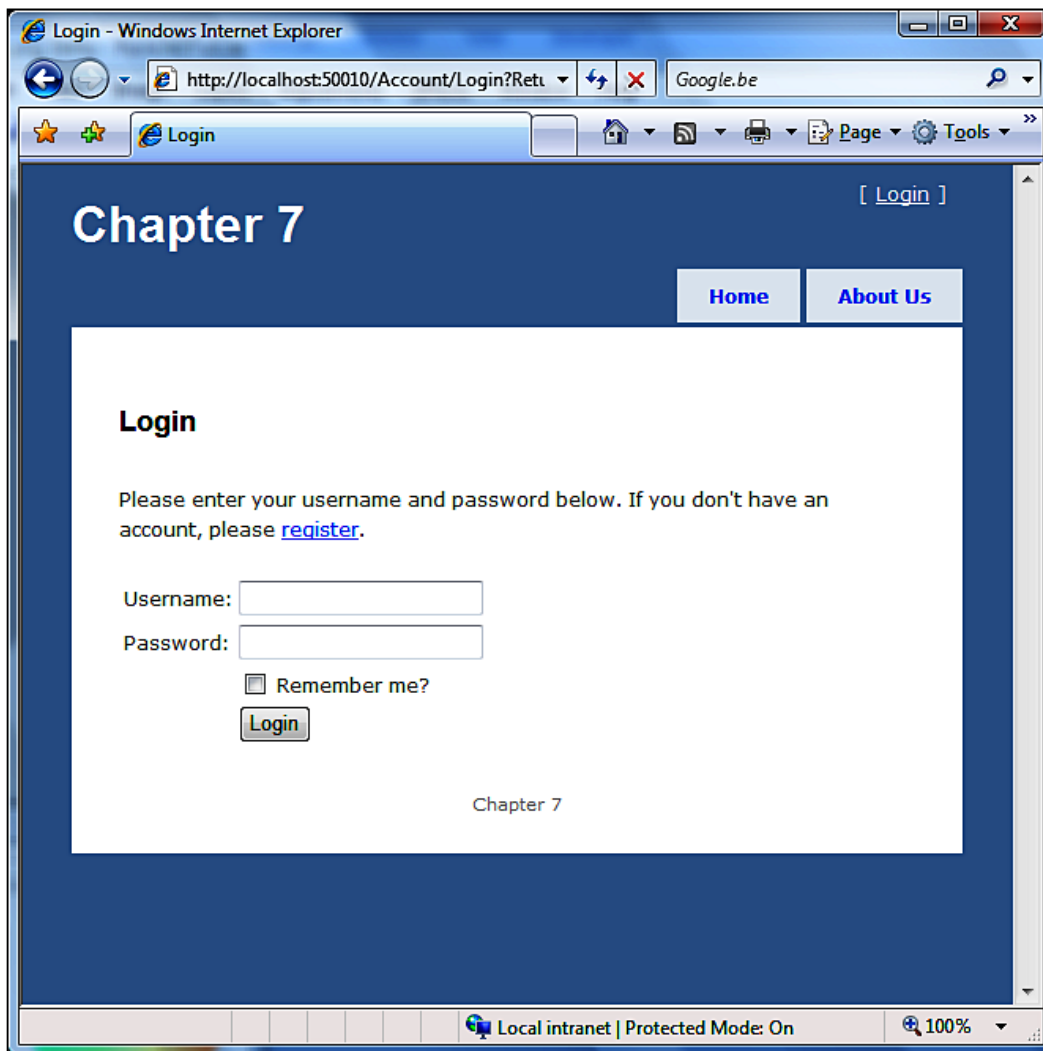
Here's a (stripped-down) example of the `web.config` file configured to redirect an unauthenticated user to `/Account/Login`:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <!-- ... -->
```

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" />
</authentication>

<!-- ... -->
</system.web>
</configuration>
```

The default login view contained in the ASP.NET MVC Web Application Visual Studio template looks like this:



Configurable authentication options

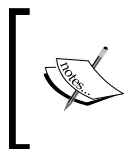
The following table lists common optional configuration directives for the `<authentication>` element. These can be modified in the application's `web.config`:

Attribute	Default	Possible values
mode	mode="Forms"	Specifies the authentication mode to be used Windows – Specifies Windows authentication as the default authentication mode; authentication responsibility is delegated to the web server (IIS) which uses Basic, Digest, Integrated Windows authentication (NTLM/Kerberos), or certificates authentication Forms – Specifies ASP.NET forms authentication as the default authentication mode Passport – Specifies Microsoft Passport as the default authentication mode None – Specifies that no authentication is required, or the application uses its own authentication mechanisms

When using forms authentication, a `<forms>` element can be nested:

Attribute	Default	Possible values
cookieless	cookieless="UseDeviceProfile"	Specifies whether cookies should be used to identify an authenticated user UseCookies – Specifies that cookies will always be used UseUri – Specifies that cookies will never be used AutoDetect – Specifies that cookies are to be used when they are supported by the user's browser; this is detected by a probing mechanism UseDeviceProfile – Specifies that cookies are to be used when specified in the ASP.NET device profile for the user's browser
defaultUrl	defaultUrl="~/Default.aspx"	The URL to redirect to after the user has been authenticated

Attribute	Default	Possible values
domain	domain=""	The domain in which the authentication cookie is valid
loginUrl	loginUrl="~/Account/Login"	The URL to redirect to if the user is required to authenticate; this is typically a login page
name	name=" .ASPXAUTH"	The name of the authentication cookie
path	path="/"	The path in which the authentication cookie is valid
protection	protection="All"	The encryption to be used for encrypting authentication cookie contents All—Specifies that the cookie is protected using data validation and encryption; this is a combination of the following two elements: Encryption—Specifies that the cookie is encrypted using DES or 3DES Validation—Specifies that the cookie is verified, to prevent possible tampered data None—No protection is applied to the authentication cookie; this is very unsecured!
requireSSL	requireSSL="false"	Specifies whether an SSL connection (HTTPS) is used to transmit the authentication cookie
timeout	timeout="30"	The timeout (in minutes) for the authentication cookie to expire
slidingExpiration	slidingExpiration="true"	Specifies whether the timeout should be reset to zero on every request



Refer to the MSDN pages at <http://msdn.microsoft.com/en-us/library/9wff0kyh.aspx> and <http://msdn.microsoft.com/en-us/library/907hb5w9.aspx> for more information on configuring authentication in ASP.NET.

Caching

When a web browser retrieves a web page, it is often cached on the local computer in the browser cache. The next time that the user requests the page, the chances are that the browser simply retrieves the local copy instead of making a new request to the web server – if the local copy is still valid. This approach increases the responsiveness of the site by requesting only the required items from the server. It also reduces the load on the web server because it is not necessary to render every page that is used by a client.

Another alternative to client-side caching is server-side caching. Imagine that a user has client-side caching enabled – the server will probably render the page only once for each user. If multiple users request a specific page, the page will be rendered for each of these users, even when each user is served with the same response.

ASP.NET offers output caching, which can be leveraged in the ASP.NET MVC framework. The ASP.NET output cache keeps a copy of a rendered page that can be returned instantly if a user requests the same page in the memory. In the ASP.NET MVC framework, output caching is enabled by decorating a controller or an action method with the `[OutputCache]` attribute.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`OutputCacheExample`).

In the following example, output caching is enabled for the `index` action method. Caching is not subject to specific settings, except that the cache should become invalid after 60 seconds.

```
[OutputCache(Duration = 60, VaryByParam="none")]
public ActionResult Index()
{
    ViewData["Title"] = "Home Page";
    ViewData["Message"] = "Welcome to ASP.NET MVC!";
    ViewData["CurrentTime"] = "The current time is " +
        DateTime.Now.ToLongTimeString();

    return View();
}
```

When using output caching, different versions of a cached result can be stored in memory, depending on things such as action method parameters, HTTP headers, and content encoding (`VaryByContentEncoding`, `VaryByCustom`, `VaryByHeader`, and `VaryByParam`).

The next example caches the rendered view for 60 seconds, after which it becomes stale (invalid). If a different parameter value for name is passed in, caching will be different. For example, if a user calls the action method with a parameter value "Maarten" for name, the cached result will be different from the cached result for the parameter value "John" for name.

```
[OutputCache(Duration = 60, VaryByParam = "name")]
public ActionResult CacheMyName(string name)
{
    ViewData["Title"] = "My cached name";
    ViewData["Message"] = "My name is " + name;
    ViewData["CurrentTime"] = "The current time is " +
        DateTime.Now.ToLongTimeString();

    return View("Index");
}
```

The following list contains all of the possible parameters for the OutputCacheAttribute:

Attribute	Description
Duration	The number of seconds to cache the page; after the specified amount of seconds, the output cache will become stale (invalid)
CacheProfile	The profile name of the cache settings to associate with the page
NoStore	Determines whether to prevent secondary storage of sensitive information
VaryByContentEncoding	A list of content encoding strings that are used to vary the output cache
VaryByCustom	A string that represents custom output caching requirements; if this attribute is given a value of browser, the cache is varied by browser name and major version information; if a custom string is entered, the GetVaryByCustomString method in the application's Global.asax file should be implemented
VaryByHeader	A list of HTTP headers to vary the output cache
VaryByParam	A list of strings used to vary the output cache; these strings correspond to the parameter names of the action method for which a different cache version should be stored; possible values include none, an asterisk (*) and any action method parameter name
SqlDependency	A string representing an SQL dependency
Location	The location where the cache is stored; possible values are Any, Client, Downstream, Server, None, and ServerAndClient

Note that in the default project created by the ASP.NET MVC Web Application Visual Studio Template, the `AccountController` has a default attribute `[OutputCache]` attribute enabled:

```
[OutputCache(Location = OutputCacheLocation.None)]
```

There might be situations where a user is able to see another user's cached content. By using the `OutputCacheLocation.None`, caching is disabled. On the `AccountController`, this is a safety measure to prevent protected content from being cached.

Also, check my blog at <http://blog.maartenballiauw.be/post/2008/07/01/Extending-ASPNET-MVC-OutputCache-ActionFilterAttribute-Adding-substitution.aspx> for an `OutputCache` attribute that also supports page substitution.

Globalization

Whenever you are creating an ASP.NET MVC web application, the chances are that you are creating it for a multilingual audience. This will be the case when building an intranet web application for a multinational company and also when developing a larger, public web site. Globalization allows your application to be used by people in different parts of the world, speaking different languages.

Globalization is a combination of localization and internationalization. Localization is the process of adapting the text and content of an application to a specific language. Internationalization is the process of displaying the application in the correct manner. For example, some languages are read left-to-right, while others are read right-to-left. ASP.NET Webforms offers globalization features that will automatically adjust formatting and languages depending on user preferences or a browser's accepted languages.

Resources

When working with globalization in ASP.NET, you will be working with resources. Resources are a collection of key-value pairs containing translations for a certain language. For example, one can have a resource file for English, which contains all of the English text used on a form. Another resource file would contain the same text, but in Dutch.

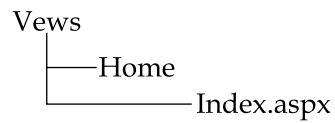
Within ASP.NET, resource files are compiled into so-called satellite assemblies. After the compilation of the `MyApplication`, there will be a `MyApplication.dll` file containing not only the application logic, but also files called `MyApplication.en-us.dll`, `MyApplication.fr-fr.dll`, `MyApplication.nl-be.dll`, and so on. These extra DLL files contain all of the translations for a specific culture, and are used by the .NET framework when the `MyApplication.dll` file is started in one of the specified languages.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`InternationalizationExample`).

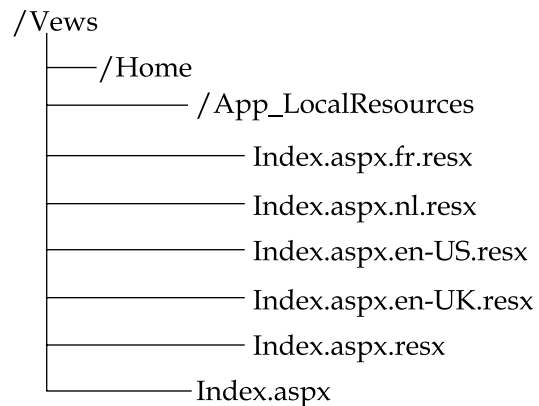
Using local resources

Local resources are resources specific to a single view or a partial view, and should be used for providing various versions in different languages. These local resources are stored in the `App_LocalResources` subfolder of the folder containing the view.

As an example, take the **Views | Home | Index.aspx** view. By default, the folder structure of this will be:



When localizing this view using local resources, folder structure will look like this:



Notice that each local resource in the `App_LocalResources` folder contains language and/or culture-specific copies named as the view, and the language and/or culture, for example, `Index.aspx.language-culture.resx`. A default local resource that will be used if no matching language/culture is found is available in `Index.aspx.resx`.

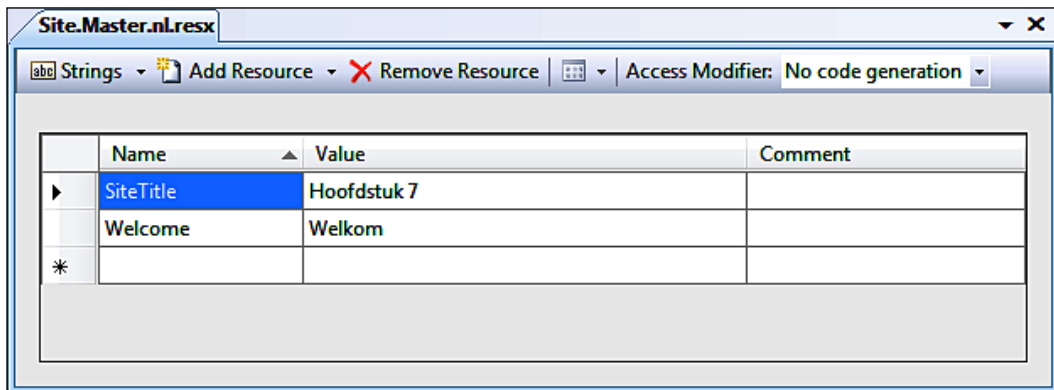
Local resources can be printed in a view by using the following syntax:

```
<%%$ Resources:SomeResourceName %>
```

For example, the local resource named `PageTitle` can be retrieved from the local resource as follows:

```
<%%$ Resources:PageTitle %>
```

The following screenshot illustrates the resource editor displaying the Dutch resource file for a master page:



We can use the resources from a resource file in a view. The main text is retrieved from a local resource file, and will display text depending on the user's language and/or culture.

```
<%@ Page Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
    AutoEventWireup="true"
    CodeBehind="Index.aspx.cs"
    Inherits="InternationalizationExample.Views.Home.Index" %>

<asp:Content ID="indexContent"
    ContentPlaceHolderID="MainContent"
    runat="server">
    <h2>
        <asp:Literal runat="server" Text="<%%$ Resources:Message %>" />
    </h2>
    <p>
        <asp:Literal runat="server" Text="<%%$ Resources:MainText %>" />
    </p>
</asp:Content>
```



The example code contains additional `HtmlHelper` extension methods to use ASP.NET MVC style code:

```
<%=Html.Resource("Message") %>
```

Using global resources

Global resources are resources that are valid for the entire web application. For example, contact email addresses, site title, and so on, can be stored as a global resource in the web application root subfolder `App_GlobalResources`.

Global resources can be printed on a view using the following syntax:

```
<%= $Resources:GlobalResourceName, SomeResourceName %>
```

For example, for retrieving the site title from a global resource named `SiteConstants.resx`, the following code can be used:

```
<%= $Resources:SiteConstants, SiteTitle %>
```



The example code contains additional `HtmlHelper` extension methods to use ASP.NET MVC style code:

```
<%=Html.Resource("SiteConstants", "SiteTitle") %>
```

Setting language and culture preferences

Each user requesting an action method in the ASP.NET MVC framework is assigned a worker thread on the web server. A worker thread is a thread that is started at a client's request. There is no guarantee that a worker thread is assigned to the same user every time—one user can be served a response by worker thread A and afterwards by worker thread D. Because language and culture are thread-specific, the culture and language settings on a thread should be repeated for each request.

Language and culture preferences can be set by passing them as request variables, storing them in a session, or using a cookie. As users often bookmark a specific page, it is recommended to provide support for passing these settings as request variables. This can be done by adding a new route to the route table, allowing both URLs in the form `/en-US/Home/Index` and URLs in the form `/Home/Index` (defaulting to `en-US`).

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "DefaultLocalized",
```

```

        "{language}-{culture}/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = "",
              language = "en", culture = "US" }
    );
    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = "" }
    );
}

```

The above code registers a new route named `DefaultLocalized`, which accepts a language and culture parameter that is used to determine the current worker thread language and culture.

Setting the current worker thread language and culture is the responsibility of a custom action filter attribute: `InternationalizationAttribute`.

```

using System.Globalization;
using System.Threading;
using System.Web.Mvc;

namespace InternationalizationExample.Filters
{
    public class InternationalizationAttribute :
        ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext
            filterContext)
        {
            string language = (string)filterContext.RouteData.
                Values["language"] ?? "en";
            string culture = (string)filterContext.RouteData.
                Values["culture"] ?? "US";

            Thread.CurrentThread.CurrentCulture = CultureInfo.
                GetCultureInfo(string.Format
                    ("{0}-{1}", language, culture));
            Thread.CurrentThread.CurrentUICulture = CultureInfo.
                GetCultureInfo(string.Format
                    ("{0}-{1}", language, culture));
        }
    }
}

```

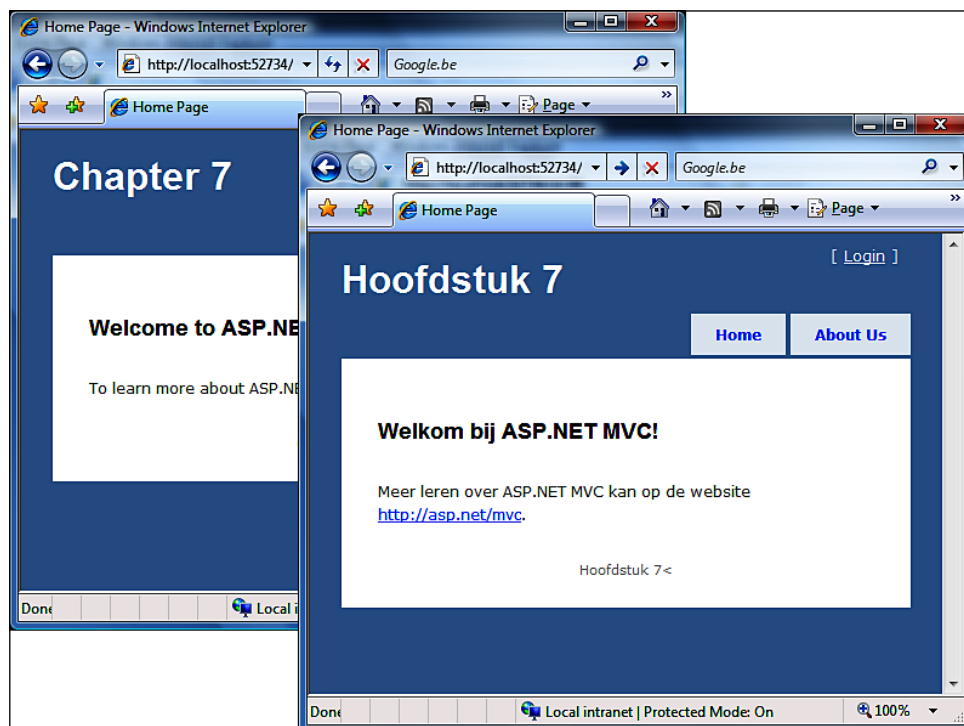
Whenever a controller action method is starting to execute, the `InternationalizationAttribute` will look for language and culture information in the route data values, which are available when a user requests a page such as `/en-UK/Home/Index`. The default language and culture used is `en (English) US (United States)`. The current thread's `CurrentCulture` and `CurrentUICulture` are set by retrieving a `CultureInfo` instance based on the language and culture variables.

The `CurrentCulture` property represents the `CultureInfo` instance that is used to format numbers, dates, and so on, whereas, the `CurrentUICulture` property represents the `CultureInfo` instance that is used to retrieve the correct local or global resource information.

Internationalizing a controller or action method can now easily be done by adding the `InternationalizationAttribute`, for example:

```
[Internationalization]
public class HomeController : Controller {
    // ...
}
```

Now, when we run the example application with different URLs, the following content is rendered:



Mixing ASP.NET Webforms and ASP.NET MVC

Not every ASP.NET MVC web application will be built from scratch. Several projects will probably end up migrating from classic ASP.NET to ASP.NET MVC. The question of how to combine both technologies in one application arises – is it possible to combine both ASP.NET Webforms and ASP.NET MVC in one web application? Luckily, the answer is *yes*.

Combining ASP.NET Webforms and ASP.NET MVC in one application is possible – in fact, it is quite easy. The reason for this is that the ASP.NET MVC framework has been built on top of ASP.NET. There's actually only one crucial difference: ASP.NET lives in `System.Web`, whereas ASP.NET MVC lives in `System.Web`, `System.Web.Routing`, `System.Web.Abstractions`, and `System.Web.Mvc`. This means that adding these assemblies as a reference in an existing ASP.NET application should give you a good start on combining the two technologies.

Another advantage of the fact that ASP.NET MVC is built on top of ASP.NET is that data can be easily shared between both of these technologies. For example, the `Session` state object is available in both the technologies, effectively enabling data to be shared via the `Session` state.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`MixingBothWorldsExample`).

Plugging ASP.NET MVC into an existing ASP.NET application

An ASP.NET Webforms application can become ASP.NET MVC enabled by following some simple steps. First of all, add a reference to the following three assemblies to your existing ASP.NET application:

- `System.Web.Routing`
- `System.Web.Abstractions`
- `System.Web.Mvc`

After adding these assembly references, the ASP.NET MVC folder structure should be created. Because the ASP.NET MVC framework is based on some conventions (for example, controllers are located in **Controllers**), these conventions should be respected. Add the folder **Controllers**, **Views**, and **Views | Shared** to your existing ASP.NET application.

The next step in enabling ASP.NET MVC in an ASP.NET Webforms application is to update the `web.config` file, with the following code:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="false">
      <assemblies>
        <add assembly="System.Core, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=B77A5C561934E089"/>
        <add assembly="System.Web.Extensions,
          Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35"/>
        <add assembly="System.Web.Abstractions,
          Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35"/>
        <add assembly="System.Web.Routing,
          Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35"/>
      </assemblies>
    </compilation>
    <pages>
      <namespaces>
        <add namespace="System.Web.Mvc"/>
        <add namespace="System.Web.Mvc.Ajax"/>
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing"/>
        <add namespace="System.Linq"/>
        <add namespace="System.Collections.Generic"/>
      </namespaces>
    </pages>
    <httpModules>
      <add name="UrlRoutingModule"
        type="System.Web.Routing.UrlRoutingModule,
          System.Web.Routing, Version=3.5.0.0,
          Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
    </httpModules>
  </system.web>
</configuration>
```

The above web configuration file contains all of the required modifications to your existing `web.config` in order to enable ASP.NET MVC support. First, the necessary assemblies are registered. Next, some default namespaces are added for the compilation of any web page. Finally, the routing engine is registered as an `HttpModule`. This will enable your application to accept ASP.NET MVC URLs and map them to a specific controller.

Note that your existing ASP.NET Webforms `web.config` should not be replaced by the above `web.config`! The configured sections should be inserted into an existing `web.config` file in order to enable ASP.NET MVC.

There's one thing left to do: configure routing. This can easily be done by adding the default ASP.NET MVC's global application class contents into an existing (or new) global application class, `Global.asax`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace MixingBothWorldsExample
{
    public class Global : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.IgnoreRoute("{resource}.aspx/{*pathInfo}");
            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index", id = "" }
            );
        }
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

This code registers a default ASP.NET MVC route, which will map any URL of the form `/Controller/Action/Id` into a controller instance and action method. There's one difference with an ASP.NET MVC application that needs to be noted – a catch-all route is defined in order to prevent a request for ASP.NET Webforms to be routed into ASP.NET MVC. This catch-all route looks like this:

```
routes.IgnoreRoute("{resource}.aspx/{*pathInfo}");
```

This is basically triggered on every request ending in `.aspx`. It tells the routing engine to ignore this request and leave it to ASP.NET Webforms to handle things.

With the ASP.NET MVC assemblies referenced, the folder structure created, and the necessary configurations in place, we can now start adding controllers and views. Add a new controller in the **Controllers** folder, for example, the following simple HomeController:

```
using System.Web.Mvc;

namespace MixingBothWorldsExample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "This is ASP.NET MVC!";
            return View();
        }
    }
}
```

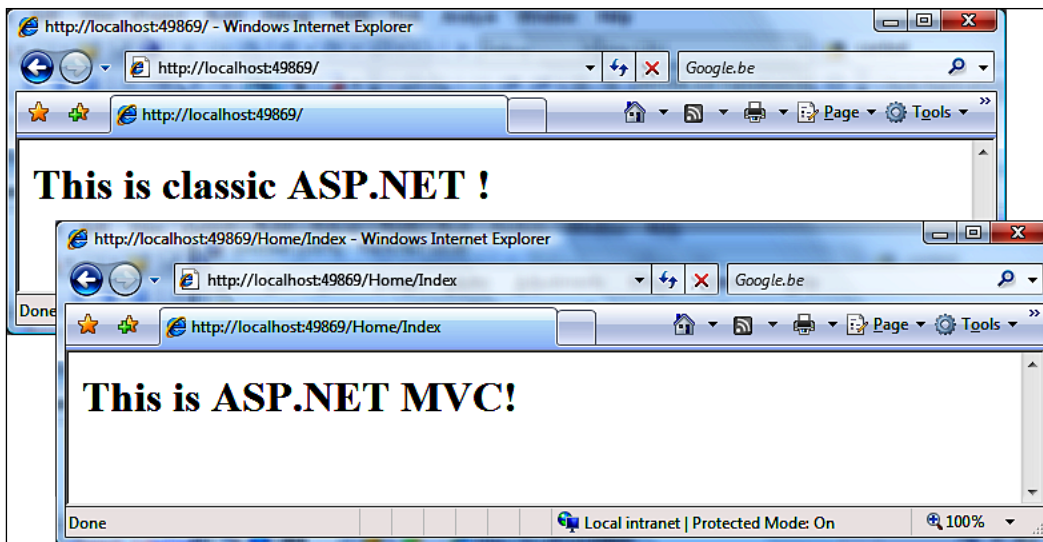
The above controller will simply render a view, and pass it a message through the ViewData dictionary. This view, located in **Views | Home | Index.aspx**, would look like this:

```
<%@ Page Language="C#"
    AutoEventWireup="true"
    CodeBehind="Index.aspx.cs"
    Inherits="MixingBothWorldsExample.Views.Home.Index" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <div>
        <h1><%=Html.Encode(ViewData["Message"]) %></h1>
    </div>
</body>
</html>
```

The above view renders a simple HTML page and renders the ViewData dictionary's message as the page title.



Plugging ASP.NET into an existing ASP.NET MVC application

The road to enabling an existing ASP.NET MVC web application to serve ASP.NET Webforms contents is actually quite easy. Because the ASP.NET MVC framework is built on top of ASP.NET Webforms, any classic web form will automatically be available from an ASP.NET MVC web application. This means that any ASPX file will be rendered using ASP.NET Webforms, unless the route table contains a matching route for handling an ASP.NET MVC request.

To avoid strange results in a mixed application, consider adding a catch-all route to the route table for ASPX pages, which will ignore any requests to ASP.NET Webforms, and will route only ASP.NET MVC requests. This can be done in the global application class, `Global.asax`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MixingBothWorldsExample
{
    public class Global : System.Web.HttpApplication
    {
    }
```

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.IgnoreRoute("{resource}.aspx/{*pathInfo}");
    routes.MapRoute(
        "Default",
// Route name
        "{controller}/{action}/{id}",
// URL with parameters
        new { controller = "Home", action = "Index", id = "" }
// Parameter defaults
    );
}

protected void Application_Start()
{
    RegisterRoutes(RouteTable.Routes);
}
}
```

The above code registers a default ASP.NET MVC route, which will map any URL in the form `/Controller/Action/Id` into a controller instance and action method. The catch-all route for ASP.NET Webforms looks like this:

```
routes.IgnoreRoute("{resource}.aspx/{*pathInfo}");
```

Sharing data between ASP.NET and ASP.NET MVC

Whether you are creating a new mixed ASP.NET Webforms-ASP.NET MVC application—or doing a migration, the chances are that you will need to share data between the two technologies. For example, a form can be posted by an ASP.NET Webforms page to an ASP.NET MVC action method.

Because the ASP.NET MVC framework is built on top of ASP.NET Webforms, the following objects are always available in both technologies:

- HttpContext
- Session
- Server
- Request
- Response
- Cookies

This way, it is easy to set a Session state item in classic ASP.NET Webforms and read it in ASP.NET MVC.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace MixingBothWorldsExample
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Session["SharedData"] = "This message is set by classic
                                   ASP.NET.";
        }
    }
}
```

The above code is a "codebehind" for a classic ASP.NET page. As you can see, it sets the SharedData dictionary item of Session to a string value. This data can easily be read easily in an ASP.NET MVC controller, for example:

```
using System.Web.Mvc;

namespace MixingBothWorldsExample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "This is ASP.NET MVC!";
            ViewData["SharedData"] = Session["SharedData"] ?? "";

            return View();
        }
    }
}
```

The Index action method tries to read data from the SharedData dictionary item of Session. If it has been set by ASP.NET Webforms or ASP.NET MVC, it is assigned to the ViewData dictionary. In other cases, an empty string is passed in.

Building views at compile time

By default, the views in ASP.NET MVC applications are built the first time that a request comes in. This means that if there is a compilation error in a view, it will only be visible the first time that a user requests for that view to be rendered. To overcome this issue, views can be compiled whenever the ASP.NET MVC application is compiled.

To build views at compile time, the following steps should be executed:

1. Open the project file in a text editor. For example, start Notepad and open the project file for your ASP.NET MVC application (that is, *MyMvcApplication.csproj*).

2. Find the top-most `<PropertyGroup>` element and add a new element named `<MvcBuildViews>`:

```
<PropertyGroup>
    ...
    <MvcBuildViews>true</MvcBuildViews>
</PropertyGroup>
```

3. Scroll down to the end of the file and uncomment the `<Target Name="AfterBuild">` element. Update its contents to match the following:

```
<Target Name="AfterBuild" Condition="'$(MvcBuildViews)'=='true' ">
    <AspNetCompiler VirtualPath="temp"
        PhysicalPath="$(ProjectDir)\..\$(ProjectName) "
    />
</Target>
```

4. Save the file and reload the project in Visual Studio.



Enabling view compilation may add some extra time to the build process. It is recommended that this is not enabled during development as a lot of compilation is typically involved during the development process.

Summary

In this chapter, we have learned how to use session state in the ASP.NET MVC framework. We've also seen that TempData is built using session state, and have looked at how to use it in an ASP.NET MVC web application.

We've also learned what membership, authentication, and authorization are, and how to configure these things in a web application. We know that we can use all existing ASP.NET membership, authentication, and authorization providers, and use them to protect access to specific controller action methods.

Another thing we have seen is output caching, and how you can make use of it in an ASP.NET MVC web application. We've also seen what internationalization is and how to use it.

We also mixed ASP.NET Webforms and ASP.NET MVC in one application and shared data between both these two technologies.

Finally, we added a post-build action to a project file to make sure that all views can be compiled.

8

AJAX and ASP.NET MVC

This chapter describes how you can use AJAX in combination with ASP.NET MVC by using ASP.NET AJAX and jQuery – the two AJAX frameworks that are widely used.

As you might know, **AJAX** is an acronym for **Asynchronous JavaScript And XML**. It is a group of web development techniques that are used for creating more responsive and rich web applications. With AJAX, data is transferred asynchronously in the background without the current page being reloaded. This asynchronous request can be created by using the `XMLHttpRequest` object, which is present in most browsers nowadays. We will see more about `XMLHttpRequest` in the following topics.

You will learn the following in this chapter:

- Which AJAX frameworks are available
- How most AJAX frameworks are built
- What **JSON (JavaScript Object Notation)** is
- How ASP.NET AJAX can be used in ASP.NET MVC web applications
- How jQuery can be used in ASP.NET MVC web applications
- How jQuery UI plugins can be used to enrich ASP.NET MVC views

Different AJAX frameworks

There are many AJAX frameworks available on the Internet. Most of them are free and they can help you to develop a dynamic web page on the client side more easily, by exposing asynchronous connections between the client and the server, and the different JavaScript classes that help you work with the **Document Object Model (DOM)**.

Many AJAX frameworks are available—ASP.NET AJAX, jQuery, Prototype, YUI, Mootools, ExtJS, and so on. In this chapter, we will be using two commonly-used AJAX frameworks—ASP.NET AJAX and jQuery.

Communication between client and server is mostly accomplished using the `XMLHttpRequest` object, which is present in most browsers. The data that is communicated can consist of HTTP, XML, or JSON. Most modern AJAX frameworks use JSON as the primary means of communicating data. This topic will cover the `XMLHttpRequest` and JSON in more detail.

XMLHttpRequest

The `XMLHttpRequest` concept was originally developed by Microsoft. When developing Outlook Web Access 2000, they needed a technology that would provide HTTP connectivity between client and server, without having to refresh a whole page in the browser.

Of course, `XMLHttpRequest` was something that only Microsoft Internet Explorer provided in those days. Luckily for we AJAX developers, the Mozilla project incorporated the first compatible native implementation of `XMLHttpRequest` in Mozilla 1.0, in 2002. Afterwards, other browsers followed this example—nowadays, Internet Explorer, Firefox, Safari, Konqueror, Opera, and so on all support a working implementation of the `XMLHttpRequest` object.

`XMLHttpRequest` can now be used by JavaScript and other web browser scripting languages to transfer XML and other text data between the client and the server. Nowadays, this is most often done using JSON, as described in the following section.

JavaScript Object Notation (JSON)

JSON is a text-based format that can be used for transmitting structured data over a network connection. Its main use is in AJAX web development, where it serves as the communication format between the browser and the web server.

A JSON string might look like this:

```
{ "Name": "Maarten", "Email": maarten@maartenballiauw.be" }
```

This translates to an object that can be used in JavaScript, and provides two properties (`Name` and `Email`) filled with data.

ASP.NET AJAX

The ASP.NET MVC framework provides out-of-the-box AJAX features that can be used in any new web application. An interesting feature is partial page updates, which is the process of refreshing only one `<div />` element's contents instead of refreshing the whole page. In order to use these features, the view (or its master page) must include two script files, `MicrosoftAjax.js` and `MicrosoftMvcAjax.js`. These files can be included by adding the following code to the `<head>` section of the master page:

```
<script type="text/javascript" src="<%=Url.Content("~/Scripts/
MicrosoftAjax.js")%>"></script>
<script type="text/javascript" src="<%=Url.Content("~/Scripts/
MicrosoftMvcAjax.js")%>"></script>
```

After adding these two script references, a set of helper methods can be used in any view.

ASP.NET MVC AJAX helper

The ASP.NET MVC AJAX helper provides three methods that enable you to create an AJAX request that behaves in a way similar to that of the ASP.NET `UpdatePanel` — data that is retrieved in the background can be displayed in an HTML `<div />` element. The following methods are available for the `AjaxHelper` object, exposed in every view:

- `ActionLink` — Generates a hyperlink to a controller action method. The response of the action method is rendered in a specified HTML element.
- `RouteLink` — Generates a hyperlink to a route. The response of this route is rendered in a specified HTML element.
- `BeginForm` — Generates a form, which is posted to a controller action method. The response of the action method is rendered in a specified HTML element.

All AJAX helper methods accept a parameter of the type, `AjaxOptions`. This class contains various properties that will configure how the AJAX request is performed:

AjaxOptions property	Description
Url	URL to which the asynchronous request should be made
Confirm	When a string is specified, the user is presented with a confirm dialog containing this string as the message
HttpMethod	The HTTP method for the AJAX request; this can be GET, POST or any other HTTP method that is supported by the browser's XMLHttpRequest object
UpdateTargetId	The target HTML element ID in which the AJAX response will be rendered; this can be used in conjunction with the InsertionMode property
InsertionMode	This enumeration defines the behavior of the UpdateTargetId property Replace - Replaces the current contents of the UpdateTargetId HTML element with the AJAX response InsertBefore - Inserts the AJAX response before the current contents of the UpdateTargetId HTML element InsertAfter - Inserts the AJAX response after the current contents of the UpdateTargetId HTML element
LoadingElementId	The HTML element ID that is displayed when performing an asynchronous request to the server
OnBegin	JavaScript function name that is called when an AJAX request is being made
OnComplete	JavaScript function name that is called when an AJAX request has been completed
OnFailure	JavaScript function name that is called when an AJAX request fails
OnSuccess	JavaScript function name that is called when an AJAX request is successful

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`AjaxHelpersExample`).

Here is an example view:

```
<asp:Content ID="indexContent"
    ContentPlaceHolderID="MainContent"
    runat="server">
    <h2><%= Html.Encode(ViewData["Message"]) %></h2>
    <p>
```

```

        To learn more about ASP.NET MVC visit <a
        href="http://asp.net/mvc" title="ASP.NET MVC
        Website">http://asp.net/mvc</a>.
    </p>
    <% using (Ajax.BeginForm("Echo", new AjaxOptions() {
        UpdateTargetId = "EchoTarget" })) {%>
        Echo the following text: <%=Html.TextBox("echo") %><input
        type="submit" value="Echo" />
    <% } %>
    <div id="EchoTarget"></div>
</asp:Content>

```

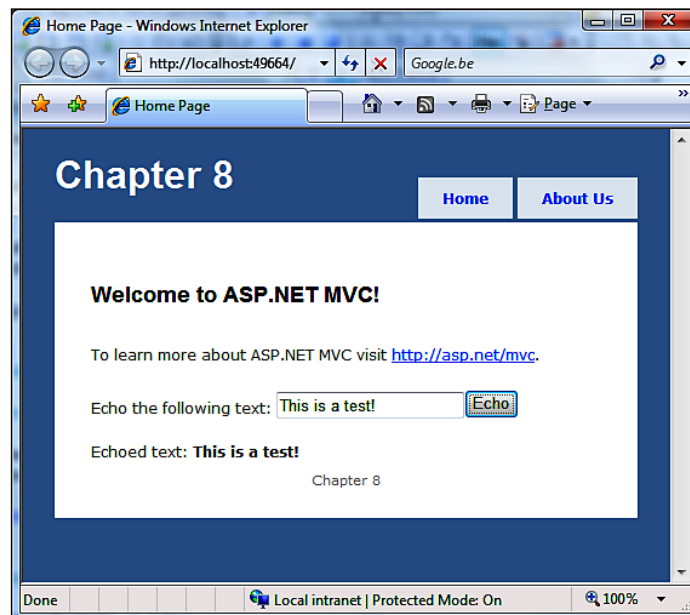
The above view contains an AJAX-enabled form, which is created by calling the `Ajax.BeginForm` method to post the contents of an echo text box to the `Echo` action method. The response of this action method will be rendered in the `EchoTarget` `<div/>` element.

```

public ActionResult Echo(string echo)
{
    ViewData["echo"] = echo;
    return View("EchoText");
}

```

The controller action method that renders the `EchoText` view looks just like a regular action method: `ViewData` is set and passed into a view. On clicking the **Echo** button, the text is rendered in a partial view. This partial view is rendered in the `<div/>` element named `EchoTarget`.



Working with JsonResult

Most action methods will return a `ViewResult` instance after processing and providing the ASP.NET MVC framework with a view name and the `ViewData` to render. When working with AJAX, a `JsonResult` instance can be returned instead of a `ViewResult` instance.

The `JsonResult` class will render an object passed in as JSON. For example, a string array containing names would look like the following JSON string:

```
["Maarten", "John", "Troy"]
```

JSON strings can be parsed by JavaScript's `eval()` function, creating a client-side object or array based on the JSON string that was generated by ASP.NET MVC's `JsonResult`.

ASP.NET AJAX is able to work with JSON in a very natural way. Take the following action method:

```
public ActionResult JsonSampleColors()
{
    string[] colors = new string[] { "Red", "Green", "Yellow",
                                     "Blue", "Orange" };

    return Json(colors);
}
```

This action method will render the following JSON string:

```
["Red", "Green", "Yellow", "Blue", "Orange"]
```

The above JSON string can be retrieved by using ASP.NET AJAX's `Sys.Net.WebRequest` class in the view. This class is a wrapper around the `XMLHttpRequest` object that we discussed earlier, and provides a set of helper methods that can be used to create a web request from JavaScript in the browser to the server.

```
<asp:Content ID="jsonSampleContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2>JsonResult Sample</h2>
    <script type="text/javascript">
        function RetrieveJsonSampleColors() {
            var request = new Sys.Net.WebRequest();
            request.set_url("/Home/JsonSampleColors");
            request.set_httpVerb("GET");
            request.add_completed(OnJsonSampleColorsCompleted);
            request.invoke();
        }
    </script>
</asp:Content>
```

```

        function OnJsonSampleColorsCompleted(executor, eventArgs) {
            if (executor.get_responseAvailable()) {
                // Should retrieve a list of colors
                var result = executor.get_object();
                // Add colors to availableColors list
                for (var i = 0; i < result.length; i++) {
                    availableColors.options[availableColors.length] =
                        new Option(result[i], result[i]);
                }
            } else {
                if (executor.get_timedOut())
                    alert("Request timeout");
                else
                    if (executor.get_aborted())
                        alert("Request aborted");
            }
        }

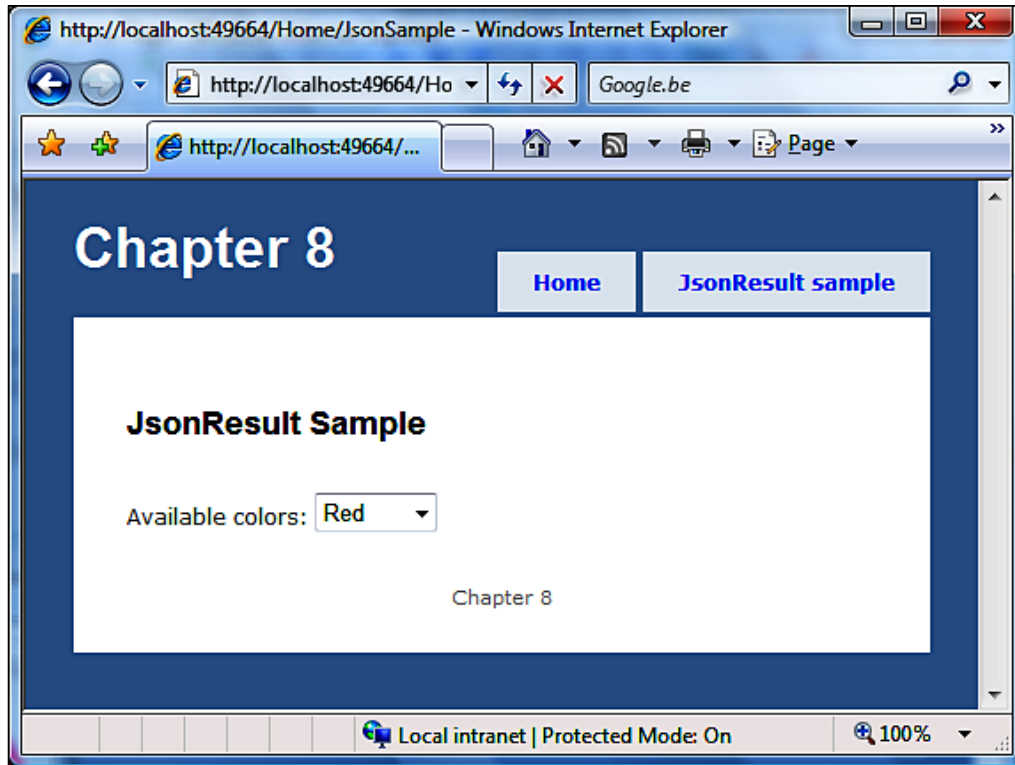
        function pageLoad() {
            RetrieveJsonSampleColors();
        }
    </script>
    <p>
        Available colors: <select id="availableColors" size="1">
                        </select>
    </p>
</asp:Content>

```

When the above view has been rendered on the client, the `RetrieveJsonSampleColors` JavaScript method is called. This method uses the `Sys.Net.WebRequest` class provided by ASP.NET AJAX to create an asynchronous request to the `/Home/JsonSampleColors` action method. It is instructed to call the `OnJsonSampleColorsCompleted` JavaScript method when a result has been received. This method distinguishes between a successful result and a timed out result, which can occur if the server is unreachable at that moment.

The `OnJsonSampleColorsCompleted` JavaScript method receives the JSON string returned by the `/Home/JsonSampleColors` action method. This result has already been parsed by the `WebRequestExecutor` class (in the `executor` parameter) and provides a `get_object()` method to access the evaluated JSON string. In this script, the evaluated JSON string is copied into the `result` variable, which is a JavaScript array containing some color strings. Each value of this array is then added to the options of the `availableColors` list-box on the HTML page.

Here's how the result looks after running the web application:



When working with large amounts of data, using AJAX in conjunction with JsonResult can decrease bandwidth usage — only data is transferred across the wire; HTML markup can be generated on the client side.

jQuery

The ASP.NET MVC framework project template contains the jQuery library, by default. jQuery is a fast JavaScript library that simplifies HTML document traversing, event handling, animating, and AJAX interactions for rapid web development. It is an alternative to Microsoft's ASP.NET AJAX, which changes the way that you write JavaScript. A task that would take 10 lines of code with traditional JavaScript can be accomplished with jQuery in just one line of code.

jQuery syntax

When first working with jQuery, the syntax may look unclear and overwhelming. This may be true the first minute you look at it. However, after playing with it for a little while, the syntax becomes clear and really useful.

In an HTML page, most elements are decorated with IDs and CSS classes that provide styling. For example, the following snippet of HTML code contains a `<div/>` element with the ID, `MainMenu`.

```
<html>
  <head>
    <title>Example of jQuery</title>
    <script type="text/javascript" src="jquery-1.2.6.js"></script>
  </head>
  <body>
    <div id="MainMenu">...</div>
  </body>
</html>
```

We can now add some jQuery to spice things up. Wouldn't it be great to have the main menu fade in when the page loads? Here's an example on how to do that, by providing jQuery with the same ids and classes that CSS uses:

```
<html>
  <head>
    <title>Example of jQuery</title>
    <script type="text/javascript" src="jquery-1.2.6.js"></script>
    <script type="text/javascript">
      $(function() {
        $("MainMenu").fadeIn();
      });
    </script>
  </head>
  <body>
    <div id="MainMenu">...</div>
  </body>
</html>
```

The above page now contains some extra JavaScript code that uses jQuery to perform some actions. The `$` is a shortcut to the jQuery object and allows access to every method that jQuery has to offer. In this case, a document ready function is registered by using the syntax `$(function() { // ... })`. This function is executed after the page has been completely loaded.

In the document ready function, the element with the ID, `MainMenu`, is searched for by using the jQuery selector API. Later, it is assigned a `fadeIn()` effect.

The jQuery selector API provides rich query features on the client side DOM document. Searching an element with a specific ID can be done using `$("#someId")`. Searching for all elements of a specific CSS class? Try `$(".someClass")`. Only need paragraphs styled with a source code CSS class? `$("p.sourceCode")` will provide the matching elements. See how this can aid you in rapid JavaScript development.

Using jQuery with ASP.NET MVC

The latest version of jQuery can be downloaded from www.jquery.com. There is also a Visual Studio documentation file available that adds full IntelliSense to the Visual Studio editor.

To make use of jQuery, add the following line of code to any view page or master page:

```
<script type="text/javascript" src="<%=Url.Content("~/Scripts/jquery-1.2.6.js")%>"></script>
```

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`jQueryExample`).

Here is an example view:

```
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2><%= Html.Encode(ViewData["Message"]) %></h2>
    <p>
        To learn more about ASP.NET MVC visit <a
        href="http://asp.net/mvc" title="ASP.NET MVC
        Website">http://asp.net/mvc</a>.
    </p>
    <script type="text/javascript">
        $(function() {
            $("#EchoForm").submit(function(e) {
                var parameters = {};
                $(this)
                    .find("input[@checked], input[@type='text'],
                    input[@type='hidden'], input[@type='password'],
                    input[@type='submit'], option[@selected], textarea")
                    .filter(":enabled")
                    .each(function() {
                        parameters[this.name || this.id ||
                        this.parentNode.name || this.parentNode.id] = this.value;
```

```

    });
    $("#EchoTarget").load($(this).attr("action"),
        parameters);
    e.preventDefault();
  });
});
</script>
<% using (Html.BeginForm("Echo", "Home", FormMethod.Get, new { id
    = "EchoForm" })) {%>
    Echo the following text: <%=Html.TextBox("echo")%><input
    type="submit" value="Echo" />
<% } %>
<div id="EchoTarget"></div>
</asp:Content>

```

The above view contains a portion of JavaScript code that uses jQuery to perform some actions. The `$` is a shortcut to the jQuery object and allows access to every method that jQuery has to offer. In this case, a document ready function is registered by using the syntax, `$(function() { // ... })`. This function is executed after the page has been completely loaded.

In the document ready function, the HTML form with the ID, `EchoForm`, is searched for, and the submit event handler is specified as a function by calling `$("#EchoForm").submit(function(e) { // ... })`.

When this form is submitted, all form fields are traversed and added to the parameters map:

```

var parameters = {};
$(this)
    .find("input[@checked], input[@type='text'], input[@type='hidden'],
    input[@type='password'], input[@type='submit'], option[@selected],
    textarea")
    .filter(":enabled")
    .each(function() {
        parameters[this.name || this.id || this.parentNode.name ||
        this.parentNode.id] = this.value;
    });

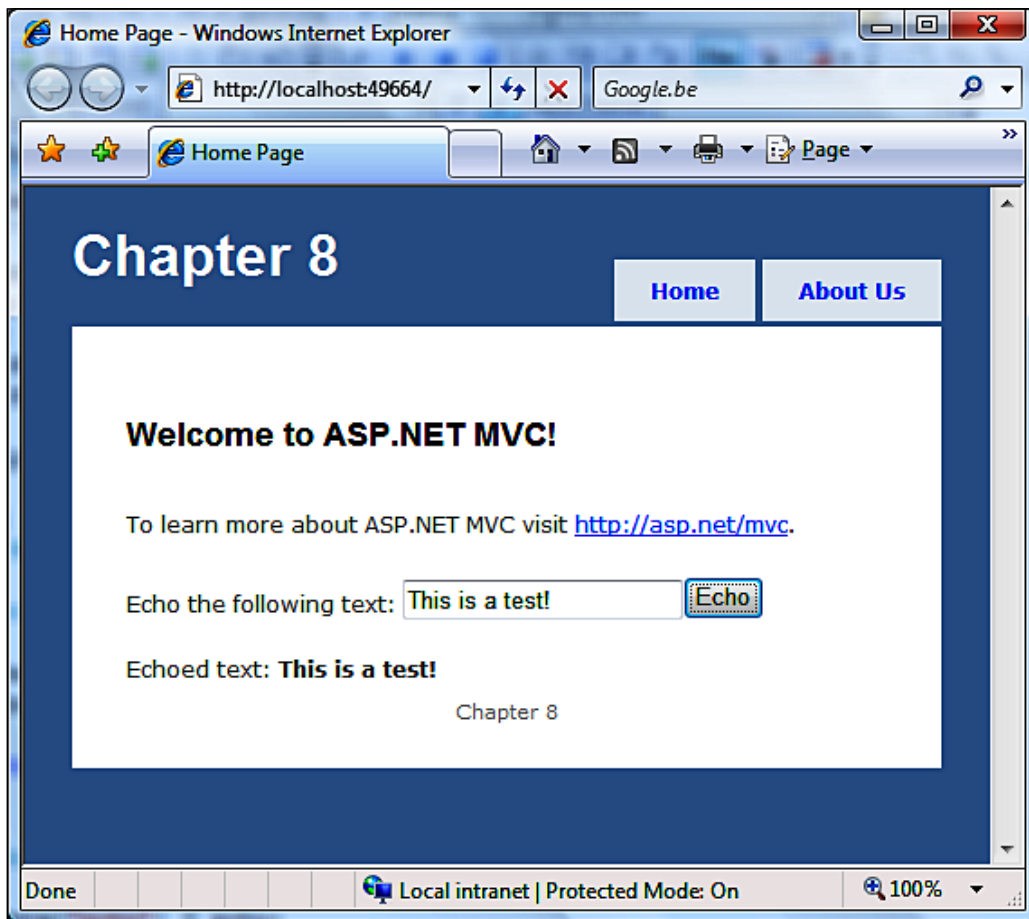
```

The jQuery object is instructed to find all input HTML elements that are enabled. For each of these elements, the names and values are added to the parameters map. Can you see the power of jQuery already? Had regular JavaScript been used, the above code snippet would consume many more lines of code than in the current case.

Finally, an AJAX request is created:

```
$("#EchoTarget").load($(this).attr("action"), parameters);  
e.preventDefault();
```

The HTML element with the ID `EchoTarget`, is instructed to load the form's action URL and pass the previously-created parameters map as parameters to the ASP.NET MVC action method. Calling `e.preventDefault()` prevents the default form submit event from being executed.



Working with JsonResult

Similar to ASP.NET AJAX, jQuery offers the possibility to parse a JSON string into a client-side JavaScript object. This JSON string can be provided by an action method, returning a `JsonResult` instead of a `ViewResult`.

For example, a string array containing names would look like the following JSON string:

```
["Maarten", "John", "Troy"]
```

jQuery is able to work with JSON in a very natural way. Take the following action method:

```
public ActionResult JsonResultSampleColors()
{
    string[] colors = new string[] { "Red", "Green", "Yellow",
                                     "Blue", "Orange" };
    return JsonResult(colors);
}
```

This action method will render the following JSON string:

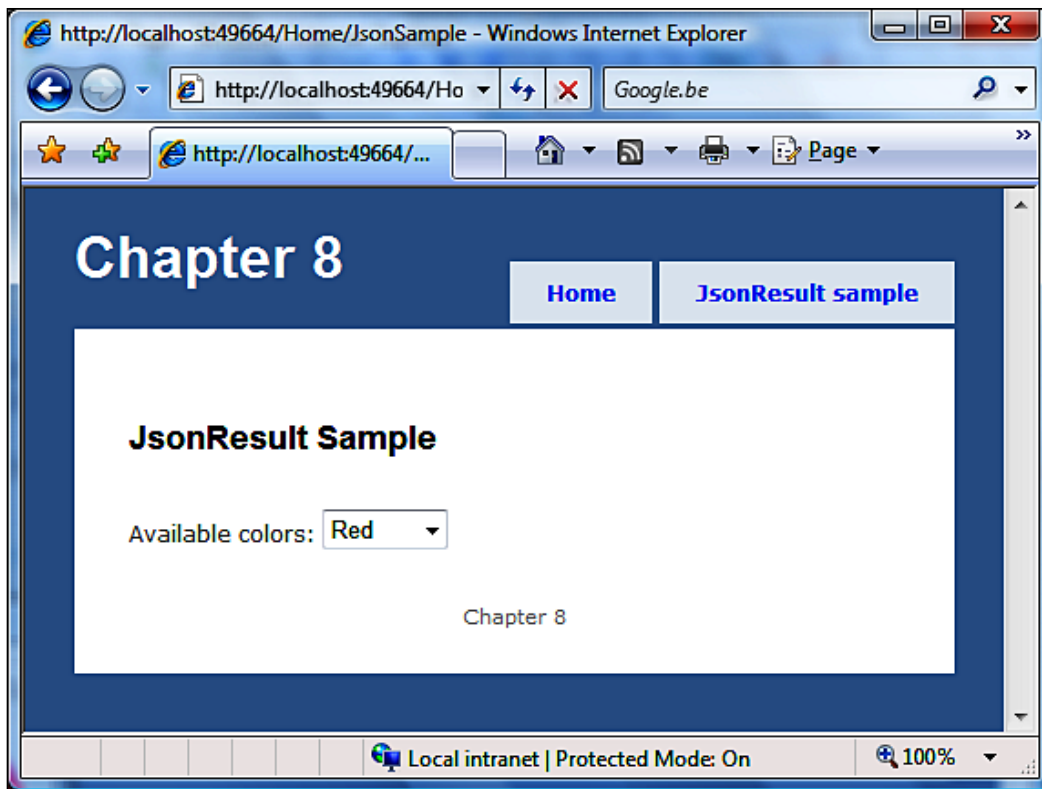
```
["Red", "Green", "Yellow", "Blue", "Orange"]
```

The above JSON string can be retrieved using jQuery's `getJSON()` method:

```
<asp:Content ID="jsonSampleContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2>JsonResult Sample</h2>
    <script type="text/javascript">
        function RetrieveJsonSampleColors() {
            $.getJSON("/Home/JsonSampleColors", function(result) {
                // Add colors to availableColors list
                for (var i = 0; i < result.length; i++) {
                    availableColors.options[availableColors.length] =
                        new Option(result[i], result[i]);
                }
            });
        }
        $(function() {
            RetrieveJsonSampleColors();
        });
    </script>
    <p>
        Available colors: <select id="availableColors"
            size="1"></select>
    </p>
</asp:Content>
```

Once the above view has been rendered on the client, the `RetrieveJsonSampleColors` JavaScript method is called from within the document ready function that jQuery provides. This method instructs jQuery to create a new asynchronous request to the `/Home/JsonSampleColors` action method, and to execute a callback function when a request has been received.

The callback function for the jQuery `getJSON()` method accepts a parameter that contains the evaluated JSON string received from the ASP.NET MVC action method. In this case, an array of colors is received and added to the options of the `availableColors` listbox on the HTML page.



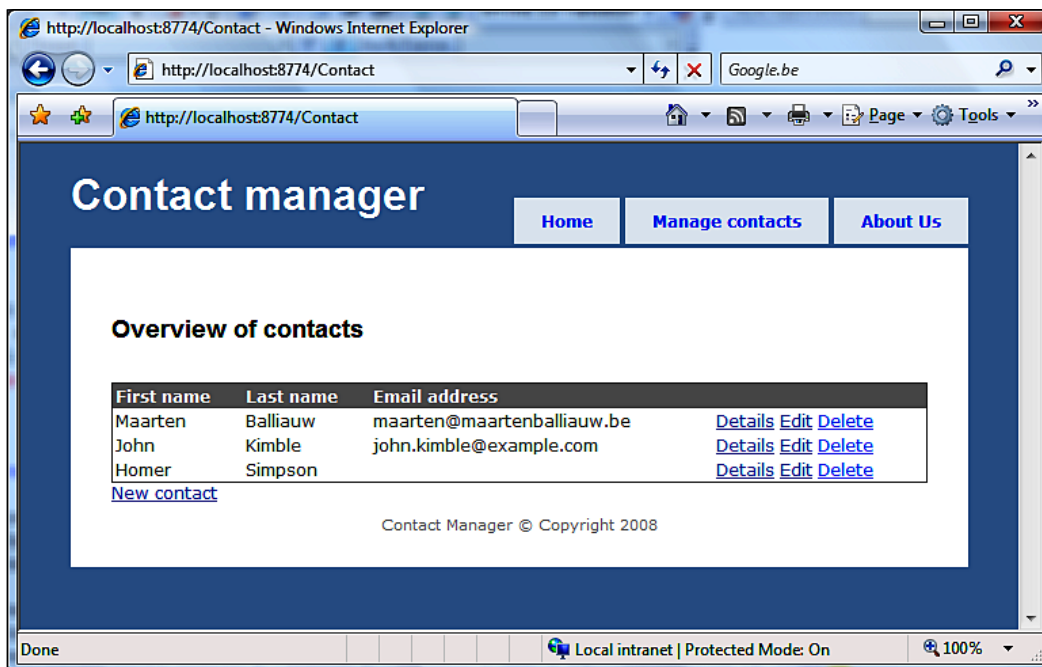
Using jQuery UI

The jQuery library offers a large set of plugins that extend the standard jQuery functionality with a lot of helpers for common actions. Plugins can be found on <http://plugins.jquery.com>.

In addition to the jQuery plugin library, a UI library is also offered, which contains popular controls such as accordion, autocomplete, color picker, date picker, dialog, magnifier, progress bar, slider, spinner, tabs, and so on. The jQuery UI library can be downloaded from <http://ui.jquery.com>.

After adding the required JavaScript, CSS, and image files to your project, the jQuery UI library can be used quite easily as an extension of the standard jQuery. The UI library is loaded whenever an action method in the web application is requested.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (ContactManagerExample). We'll extend the following screen, and show contact details with some jQuery UI elements:



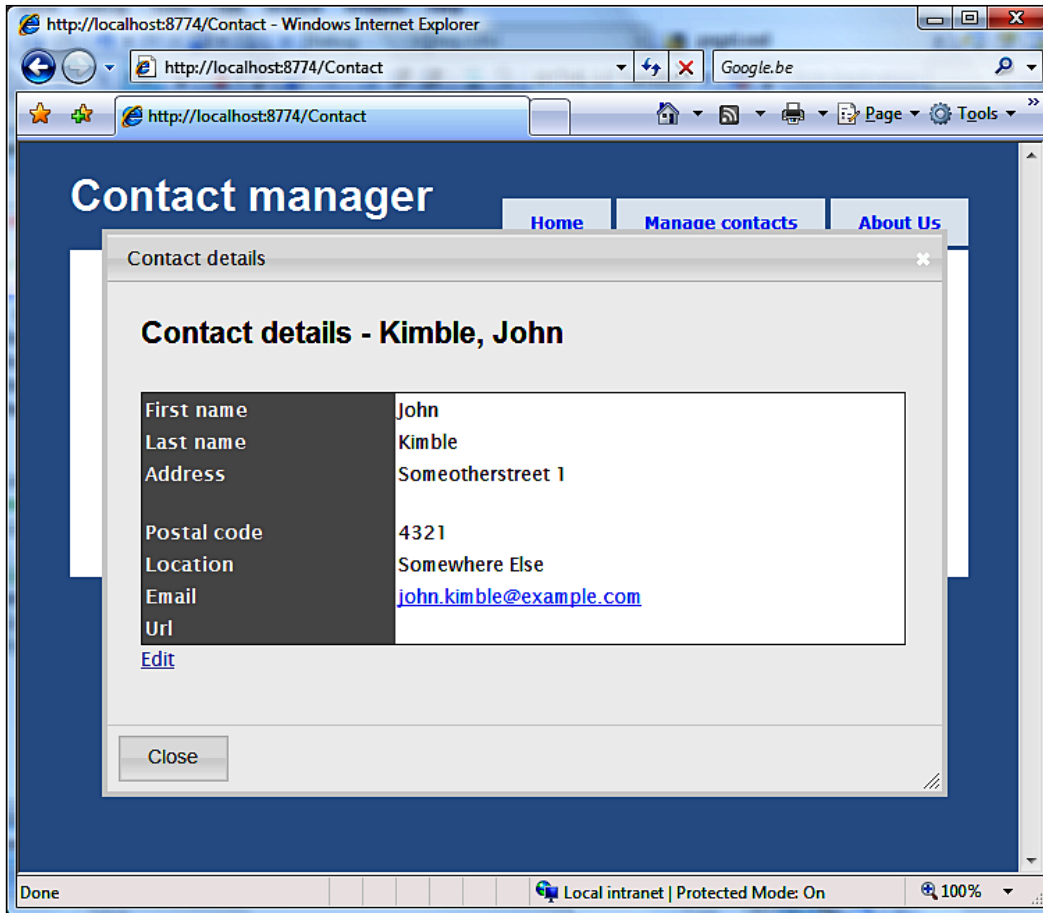
The following code can be used to show a contact's details in a jQuery UI dialog:

```
<script type="text/javascript">
    $(function() {
        // Find details links
        var detailsLinks = $("#ContactList >
            ").find("a:contains(Details)");

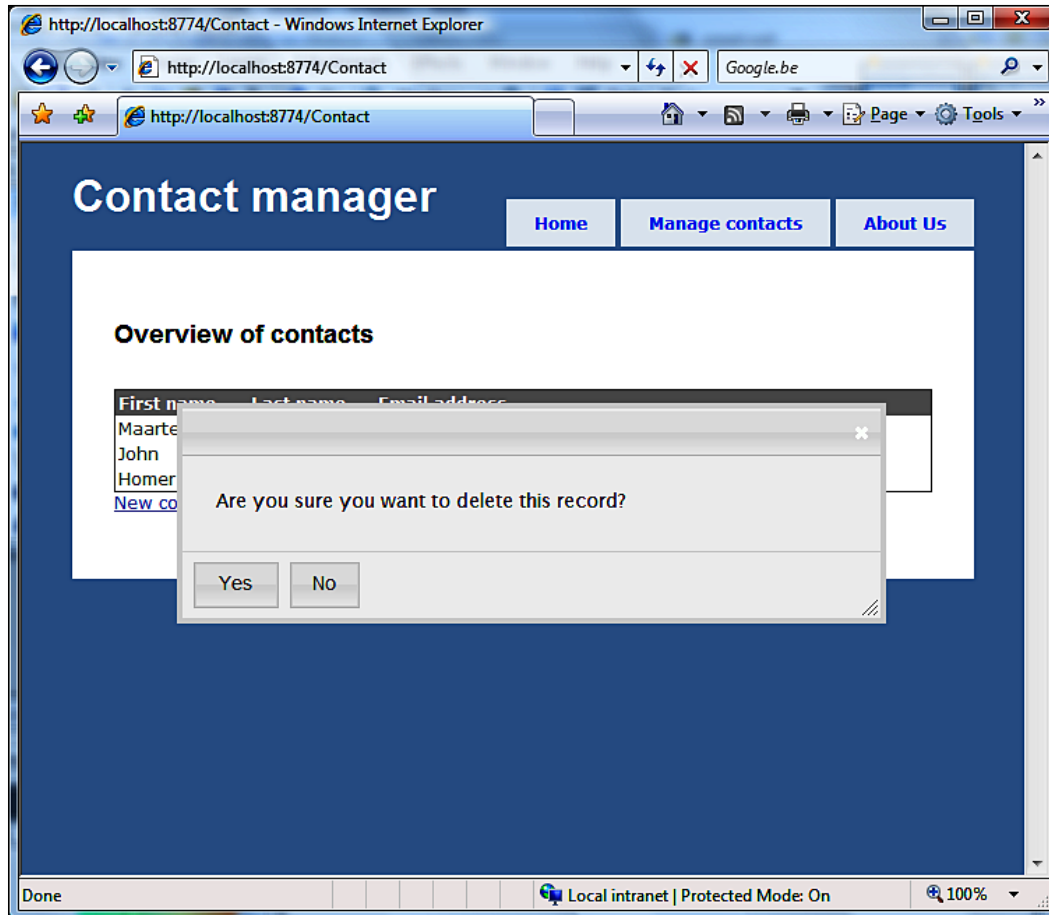
        // Set click events
        $.each(detailsLinks, function(i, val) {
            $(val).click(function(e) {
                $("<div>").load(val.href).dialog({
                    title: "Contact details",
                    width: 600,
                    height: 400,
                    open: function() {
                        $(this).parents(".ui-dialog:first > ")
                            .find(".ui-dialog-content").css('width',
                                '550px');
                    },
                    buttons: { "Close": function() {
                        $(this).dialog("close"); } }
                }).show();
                e.preventDefault();
            });
        });
    });
</script>
```

The above code snippet is executed when the page has been completely loaded. It searches for all hyperlinks that contain the text `Details`. This query can easily be executed by doing a `find()` of all `a` elements that match this criteria. Each of these `Details` hyperlinks is registered for a `click()` event handler. This event handler creates a new `<div/>` element by simply calling this in jQuery by using `$("<div>")`. The `div` element is instructed to load the hyperlink's `href` value and, after that is done, to display itself as a dialog. Some options are passed into this dialog, for example, a title for the title bar of the dialog, the default width and height, and so on. This dialog also executes some code on opening—the table element inside is rendered with a fixed width to make sure that the dialog fits on the screen.

After the dialog is shown using the `show()` method, the `e.preventDefault()` is called to make sure that the regular behavior of the hyperlink is cancelled – that is, the hyperlink is not allowed to navigate to a new page when it's already open in a jQuery dialog.



Using a similar technique, a nice looking confirmation dialog can be displayed whenever a user tries to delete a contact. This dialog asks the user if he or she is sure that he or she wants to delete the current contact and then executes the deletion when the user confirms.



Whenever the page finishes loading, the following JavaScript code is executed:

```
<script type="text/javascript">
    $(function() {
        // Find delete links
        var deleteLinks = $("#ContactList >").
            find("a:contains(Delete)");

        // Set click events
        $.each(deleteLinks, function(i, val) {
            $(val).click(function(e) {
```

```

$("<div>Are you sure you want to delete this
record?</div>").dialog({
    width: 500,
    height: 150,
    buttons: {
        "Yes": function() {
            $(this).dialog("close");
            $.getJSON(val.href, function(data) {
                if (data == true) {
                    $(val).parent().parent().find("td").
                        fadeOut('slow', function() {
                            $(this).remove();
                        });
                }
            });
        },
        "No": function() { $(this).dialog("close"); }
    }
}).show();

e.preventDefault();
});
});
</script>

```

This code snippet searches for all hyperlinks, containing the text, **Delete**, by doing a `find()` of all `a` elements that match this criteria. Each of these hyperlinks is registered for a `click()` event handler. When a user clicks on the **Delete** link, a dialog is displayed asking the user for a delete confirmation. Also, **Yes** and **No** buttons are registered to make the dialog accept one of these two choices. The **No** button simply closes the current dialog, while the **Yes** button performs the actual delete action.

When clicking on the **Yes** button, the current dialog is closed. In the background, an asynchronous call is made to the hyperlink's `href`, expecting a JSON result. If this JSON result is `true`—that is, when the ASP.NET web application has successfully deleted the record—the hyperlink's table cells are animated using a fade out, after which, the record is removed from the UI. This behavior creates a rich user experience, as the user actually watches the record fade out.

Summary

In this chapter, we have learned which AJAX frameworks are available, and how most AJAX frameworks are built. We've also learned what JSON is.

We've also seen how to use ASP.NET AJAX in an ASP.NET MVC web application, and how we can leverage the `AjaxHelper` class to aid AJAX development.

Finally, we have seen how jQuery can be used in ASP.NET MVC web applications, and how the jQuery UI plugins can be used to enrich ASP.NET MVC views.

9

Testing an Application

One of the differences between ASP.NET MVC and ASP.NET Webforms is that the ASP.NET MVC framework is easier to test than the ASP.NET Webforms framework. This is because of the fact that ASP.NET MVC has been designed with testability in mind. But even with every feature of ASP.NET MVC designed with testability in mind, there are certain aspects of classic ASP.NET that are difficult to use in tests. For example, each object in the `HttpContext` is populated by the ASP.NET runtime: `Request`, `Response`, `User`, `Cookies`, `Session`, and so on. Because unit testing an ASP.NET MVC application is possible, it should also be possible to unit test controller actions without requiring the ASP.NET runtime to be active. Fortunately, the ASP.NET MVC framework provides some interfaces and base classes that can easily be mocked. We will see more on mocking later in this chapter.

You will learn the following in this chapter:

- Some aspects of unit testing, and how to use them in an ASP.NET MVC web application
- What unit testing is, and what its advantages are
- The different unit testing frameworks available
- How to generate unit tests
- How to test an action method
- What a mocking framework is, and how it can facilitate unit testing
- How to carry out a unit test using a mocking framework
- Test routes and model updates using a mocking framework

Unit testing

In a lot of software development teams, testing is something that often takes a back seat. Even if done, a lot of teams think they are testing when one or two users click through the application and make sure that most visible errors are removed.

This cannot be considered testing. Testing should aim at covering all of the application logic, that is, every single method should be executed at least once to ensure that all code works correctly. This goal can be achieved through unit testing, which can also be pronounced as "you-need testing".

When unit-testing, you always test the smallest piece of testable software in the application—that is, every method in the application. These code portions should be tested in isolation from the rest of the code and from other tests, as other code or tests may influence the result of the test that is being executed. Also, unit tests must be repeatable—it should be possible to execute them at any given time.

The benefits of unit testing are that unit testing facilitates change. If a test is failing after some code is added to the application, it means that the application will not work as expected. One can also write a test for a new feature first, and create the code in the application later. This is called **Test-Driven Development (TDD)**.

Another benefit of unit testing is that all tests are some sort of documentation of the methods in an application, because each unit test covers exactly one method, reading the unit test can help us understand what a specific method does.

Unit testing frameworks

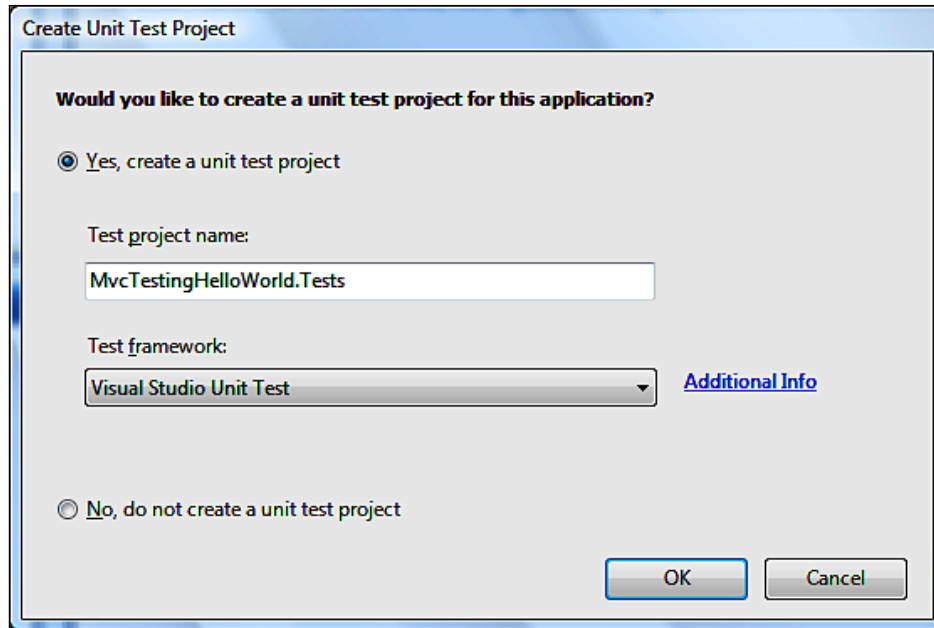
Writing unit tests is done mostly by leveraging a unit testing framework, which helps us to write unit tests and assists us in gathering test results. On the Internet, a lot of unit testing frameworks are available: MS Test (included in most Visual Studio versions), NUnit, xUnit, MbUnit, TestDriven.NET, and so on. A list of all testing frameworks can be found on Wikipedia: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

In this chapter, we will be using MS Test as the unit testing framework because it is readily available in most Visual Studio versions. Note that testing ASP.NET MVC web application can be done perfectly well using another testing framework.

Hello, unit testing!

Let's start with a sample ASP.NET MVC application. The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (MvcTestingHelloWorld).

When creating a new ASP.NET MVC application using the Visual Studio ASP.NET MVC project template, Visual Studio will ask you if you want to create a test project. This dialog box offers the choice between several unit testing frameworks that can be used for testing your ASP.NET MVC application.



In the **Create Unit Test Project** dialog box, make sure that you create a unit test project. Confirm the dialog by clicking on the **OK** button.

After the project has been created, locate the test project's `HomeControllerTest` class. You'll notice that the project template has already created two sample unit tests for the `HomeController` action methods. Notice the `[TestMethod]` attribute, which is used by Visual Studio to determine which methods should be executing when running tests.

```
[TestMethod]
public void Index()
{
    // Setup
    HomeController controller = new HomeController();

    // Execute
    ViewResult result = controller.Index() as ViewResult;

    // Verify
```

```
        ViewDataDictionary viewData = result.ViewData;
        Assert.AreEqual("Home Page", viewData["Title"]);
        Assert.AreEqual("Welcome to ASP.NET MVC!", viewData["Message"]);
    }
}
```

The above code is the test method that tests the Index action method of the HomeController class. As you can see, each test should consist of three parts: setup, execution, and verification. In the setup stage, all necessary classes are instantiated. In the execution stage, the required settings are made, and the method that is being tested is called. In the verification part, some assertions are made on the resulting data. In this case, there's an assertion that requires the Index action method of the HomeController class to return a ViewData instance containing a title, **Home Page**, and a description, **Welcome to ASP.NET MVC**.

Generating unit tests

When browsing the current ASP.NET MVC, note that there are no unit tests defined for the AccountController. The following code is the Login action method in the AccountController:

```
public ActionResult Login(string username, string password, bool?
    rememberMe)
{
    ViewData["Title"] = "Login";
    // Non-POST requests should just display the Login form
    if (Request.HttpMethod != "POST")
    {
        return View();
    }
    // Basic parameter validation
    List<string> errors = new List<string>();
    if (String.IsNullOrEmpty(username))
    {
        errors.Add("You must specify a username.");
    }
    if (errors.Count == 0)
    {
        // Attempt to login
        bool loginSuccessful = Provider.ValidateUser(username,
            password);
        if (loginSuccessful)
        {
            FormsAuth.SetAuthCookie(username, rememberMe ?? false);
            return RedirectToAction("Index", "Home");
        }
    }
}
```

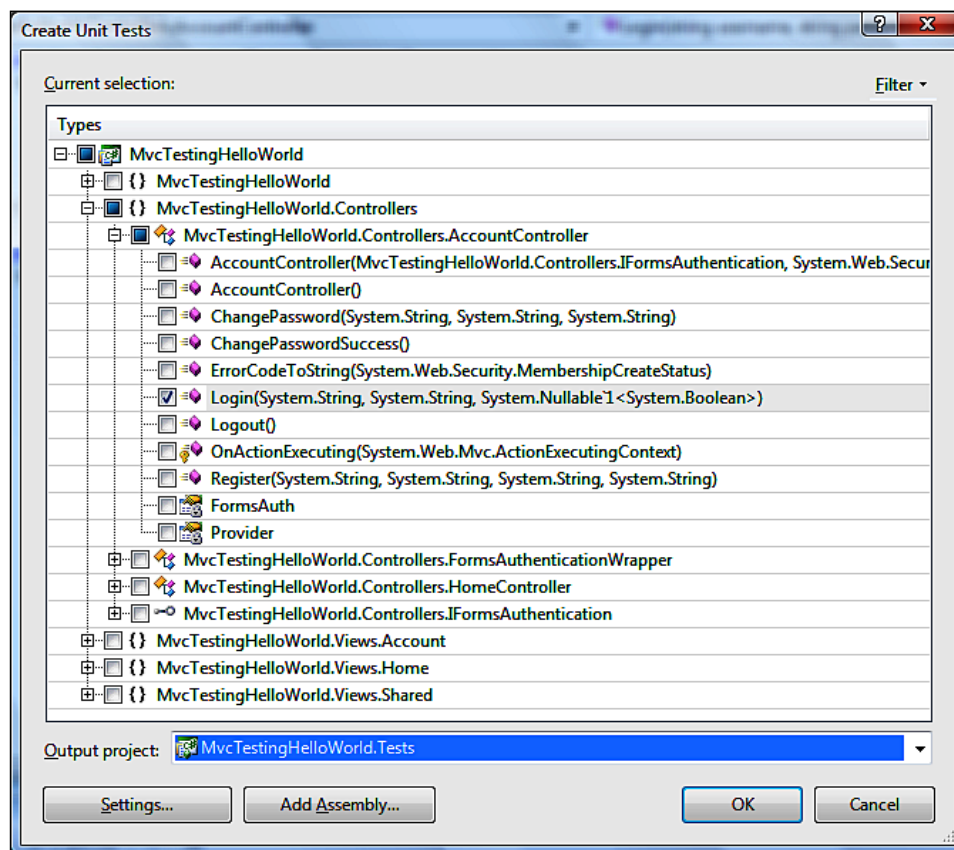
```

    }
    else
    {
        errors.Add("The username or password provided is
                    incorrect.");
    }
}

// If we got this far, something failed, redisplay form
ViewData["errors"] = errors;
ViewData["username"] = username;
return View();
}

```

To create a unit test for this method, locate the `AccountController` and find the `Login` action method. Right-click on this method and pick the menu item **Create Unit Tests....** A dialog box will be displayed, which can be confirmed by click on the **OK** button.



After the **Create Unit Tests** wizard has been completed, a unit test class is generated, based on the actual code in your ASP.NET MVC application. You can remove the following attributes from the test code:

```
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("...\MvcTestingHelloWorld", "/")]
[UrlToTest("http://localhost:50954/")]
```

The above attributes would instruct the unit testing engine to fire up the ASP.NET development server and test a specific URL. Because the ASP.NET MVC team stated that this is not required, and that the tests can be executed without the need for a web server, these attributes can be safely removed. In fact, not removing them would slow down unit testing because the ASP.NET development server would be started for each run.

Testing the action method

Let's test the following scenario. When a user provides username and password, the `AccountController` is expected to verify the credentials and authenticate the user.

Open the `AccountController` and have a look at the constructor.

```
public AccountController()
    : this(null, null)
{
}

public AccountController(IFormsAuthentication formsAuth,
    MembershipProvider provider)
{
    FormsAuth = formsAuth ?? new FormsAuthenticationWrapper();
    Provider = provider ?? Membership.Provider;
}
```

Note that the `AccountController` constructor has two overloads: one is a parameterless constructor that defaults to the standard ASP.NET membership provider, and the other is an overload that accepts an `IFormsAuthentication` instance and a `MembershipProvider` instance. Unfortunately, both default to using cookies and a real database server. Because we do not want to use the ASP.NET development server, the chances are that we may not want to use a real database server, either.

In order to do this, we'll have to provide an `IFormsAuthentication` implementation and a `MembershipProvider` instance to the constructor. The default implementations are not an option because they rely on a real database to be available. Instead, we can create our own implementations based on the `IFormsAuthentication` interface, which defines how a cookie can be set, and a user can be logged out. ASP.NET's `MembershipProvider`, a base class that provides all of the methods, is required to work with users.

```
public interface IFormsAuthentication
{
    void SetAuthCookie(string userName, bool createPersistentCookie);
    void SignOut();
}

public abstract class MembershipProvider : ProviderBase
{
    protected MembershipProvider();
    public abstract string ApplicationName { get; set; }
    public abstract bool EnablePasswordReset { get; }
    public abstract bool EnablePasswordRetrieval { get; }
    public abstract int MaxInvalidPasswordAttempts { get; }
    public abstract int MinRequiredNonAlphanumericCharacters { get; }
    public abstract int MinRequiredPasswordLength { get; }
    public abstract int PasswordAttemptWindow { get; }
    public abstract MembershipPasswordFormat PasswordFormat { get; }
    public abstract string PasswordStrengthRegularExpression { get; }
    public abstract bool RequiresQuestionAndAnswer { get; }
    public abstract bool RequiresUniqueEmail { get; }
    public event MembershipValidatePasswordEventHandler
        ValidatingPassword;
    public abstract bool ChangePassword(string username, string
        oldPassword, string newPassword);
    public abstract bool ChangePasswordQuestionAndAnswer(string
        username, string password, string newPasswordQuestion, string
        newPasswordAnswer);
    public abstract MembershipUser CreateUser(string username, string
        password, string email, string passwordQuestion, string
        passwordAnswer, bool isApproved, object providerUserKey, out
        MembershipCreateStatus status);
    protected virtual byte[] DecryptPassword(byte[] encodedPassword);
    public abstract bool DeleteUser(string username, bool
        deleteAllRelatedData);
    protected virtual byte[] EncryptPassword(byte[] password);
    public abstract MembershipUserCollection FindUsersByEmail(string
        emailToMatch, int pageIndex, int pageSize, out int
```

```
        totalRecords);  
public abstract MembershipUserCollection FindUsersByName(string  
    usernameToMatch, int pageIndex, int pageSize, out int  
    totalRecords);  
public abstract MembershipUserCollection GetAllUsers(int  
    pageIndex, int pageSize, out int totalRecords);  
public abstract int GetNumberOfUsersOnline();  
public abstract string GetPassword(string username, string  
    answer);  
public abstract MembershipUser GetUser(object providerUserKey,  
    bool userIsOnline);  
public abstract MembershipUser GetUser(string username, bool  
    userIsOnline);  
public abstract string GetUserNameByEmail(string email);  
protected virtual void OnValidatingPassword  
    (ValidatePasswordEventArgs e);  
public abstract string ResetPassword(string username, string  
    answer);  
public abstract bool UnlockUser(string userName);  
public abstract void UpdateUser(MembershipUser user);  
public abstract bool ValidateUser(string username, string  
    password);  
}
```

Implementing the `IFormsAuthentication` would not be difficult. However, implementing a `MembershipProvider` seems like a tedious job. This is where mocking frameworks come into play. A mocking framework will generate a fake implementation, on which you can define only those methods that are required to run your tests.

Mocking frameworks

Mocking frameworks allow you to easily create "fake" instances of classes in an application. These "fakes" or mocks can be used in unit testing. When searching the Internet, many mocking frameworks are available: Moq, Rhino Mocks, TypeMock, EasyMock, and so on. Each mocking framework does the same thing, but in its own manner—creating fake class implementations on which you have to write only the code that is actually required for your unit tests to run.

In this book, we will use Moq (<http://code.google.com/p/moq/>). Moq is an open source mocking framework that provides the easiest interface for creating mock objects. Other frameworks, such as TypeMock, have other advantages, but Moq will be sufficient for most cases.

The example in this topic is based on an ASP.NET MVC web application that can be found in the sample code for this book (MockingExample).

Consider the following source code:

```
public interface ICalculator
{
    int Add(int a, int b);
    int Subtract(int a, int b);
}

public class CalculationEngine
{
    public ICalculator Calculator { get; set; }
    public CalculationEngine(ICalculator calculator) {
        this.Calculator = calculator;
    }
    public int AddMultiple(int[] numbers)
    {
        int result = 0;
        foreach (int number in numbers)
        {
            result = Calculator.Add(result, number);
        }
        return result;
    }
}
```

The above source code is a simple example of an application. There's an `ICalculator` interface, which is used by a `CalculationEngine` that adds multiple numbers and returns the result. This `AddMultiple` method internally uses the `ICalculator` implementation that was passed in through the `CalculationEngine` constructor.

Let's create a test for the `AddMultiple` method, which adds multiple integers: 2 and 3. The expected result would be 5. For this, we only want to test `CalculationEngine`, and not `ICalculator`. Instead, we will expect that `ICalculator` returns some predefined values.

```
/// <summary>
///A test for AddMultiple
///</summary>
[TestMethod()]
public void AddMultipleTest()
{
```

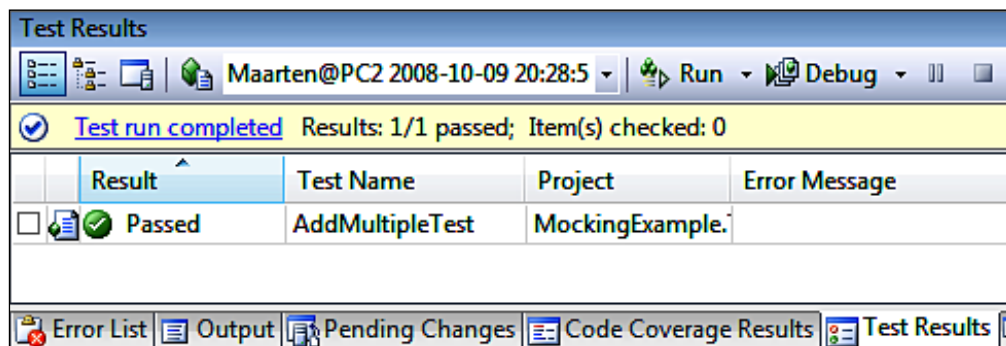
```
var calculatorMock = new Mock<ICalculator>();
calculatorMock.Expect(c => c.Add(0, 2)).Returns(2);
calculatorMock.Expect(c => c.Add(2, 3)).Returns(5);

CalculationEngine target = new CalculationEngine
                           (calculatorMock.Object);

int[] numbers = new int[] { 2, 3 };
int expected = 5;
int actual = target.AddMultiple(numbers);
Assert.AreEqual(expected, actual);
}
```

In the above test method, a mock for `ICalculator` is first created. It is expected to return 2 when the `Add` method is passed 0 and 2. When it's passed 2 and 3, it's expected to return 5. No implementation of the `ICalculator` interface is required – the mocking framework creates the implementation in-memory, and applies the expected behavior.

The `calculatorMock.Object` can be passed into the `CalculationEngine` constructor, which will now execute a test with the numbers, 2 and 3. As expected, the test will pass because the mocking object has been instructed to accept the requested additions.



Testing the Login action method

Back in the `Login` action method of `AccountController`, when a user provides the username and password, the `AccountController` is expected to verify the credentials and authenticate the user.

Let's start with setting up the mock objects:

```
var formsAuthenticationMock = new Mock<IFormsAuthentication>();
formsAuthenticationMock.Expect(f => f.SetAuthCookie(
    "testlogin", false)).AtMostOnce();

var membershipMock = new Mock<MembershipProvider>();
membershipMock.Expect(m => m.ValidateUser(
    "testlogin", "testpassword")).Returns(true);
```

Both the `IFormsAuthentication` interface and the `MembershipProvider` abstract class are mocked. The `IFormsAuthentication` mock is instructed to expect a call to `SetAuthCookie` with parameters `testlogin` and `false`, once at the most. The `MembershipProvider` mock is instructed to return `true` if a user with the username `testlogin` and the password `testpassword` tries to log in. Isn't this much faster than creating a full implementation of the `IFormsAuthentication` interface and the `MembershipProvider` class?

The next thing to do is to set up the `AccountController`.

```
AccountController target = new AccountController(
    formsAuthenticationMock.Object, membershipMock.Object);
target.SetFakeControllerContext();
target.Request.SetHttpMethodResult("POST");
```

The `AccountController` is instantiated with both of the mock objects being passed in via the constructor. Also, a fake `HttpContext` is created (see the section on *Mocking ASP.NET components*).

The crucial part of each unit test is the actual execution of the action method. If everything goes correctly, it should return a `RedirectToRouteResult`. Successful logins are redirected to the `Index` action method of the `HomeController` class.

```
RedirectToRouteResult actual = target.Login(
    "testlogin", "testpassword", false) as RedirectToRouteResult;
```

Now, there's only one thing left to do—perform the necessary assertions. Make sure to assert the resulting controller (should be `Home`) and action (should be `Index`). Also, as we instructed the mocking framework to expect a call, verify that the calls into the `formsAuthenticationMock` have been made.

```
Assert.AreEqual("Home", actual.RouteValues["controller"]);
Assert.AreEqual("Index", actual.RouteValues["action"]);
formsAuthenticationMock.Verify();
```

The full test code for the `Login` action method of the `AccountController` class scenario is as follows:

```
/// <summary>
///A test for Login
///</summary>
[TestMethod()]
public void LoginTest()
{
    // Setup mocks
    var formsAuthenticationMock = new Mock<IFormsAuthentication>();
    formsAuthenticationMock.Expect(f => f.SetAuthCookie(
        "testlogin", false)).AtMostOnce();

    var membershipMock = new Mock<MembershipProvider>();
    membershipMock.Expect(m => m.ValidateUser(
        "testlogin", "testpassword")).Returns(true);

    // Setup controller
    AccountController target = new AccountController(
        formsAuthenticationMock.Object, membershipMock.Object);
    target.SetFakeControllerContext();
    target.Request.SetHttpMethodResult("POST");

    // Execute
    RedirectToRouteResult actual = target.Login(
        "testlogin", "testpassword", false) as RedirectToRouteResult;

    // Verify
    Assert.AreEqual("Home", actual.RouteValues["controller"]);
    Assert.AreEqual("Index", actual.RouteValues["action"]);
    formsAuthenticationMock.Verify();
}
```

Mocking ASP.NET components

When unit-testing ASP.NET MVC applications, ASP.NET components such as `Request` and `Response` are often mocked. These components are normally filled by the ASP.NET runtime, which is unavailable when performing unit tests. To make use of `HttpContext`, `Request`, `Response`, `SessionState` and server variables, use the `MvcMockHelpers` extension methods, which are listed in Appendix B, instead of firing up a web server to perform the tests.

Whatever mocking framework you are using (Moq, Rhino Mocks, or TypeMock), these `MvcMockHelpers` will provide you with some extension methods and utility functions which mock the ASP.NET internals, which any ASP.NET MVC application relies upon.

The following table lists the (extension) methods that are available in the `MvcMockHelpers` class:

Class	(Extension) method	Description
<code>MvcMockHelpers</code>	<code>FakeHttpContext</code>	Creates a mock <code>HttpContextBase</code> containing HTTP context, session, request and response.
<pre>// Fake HttpContext as if it was http://www.mysite.com/Home/Index being requested HttpContextBase httpContext = MvcMockHelpers.FakeHttpContext ("http://www.mysite.com/Home/Index");</pre>		
<code>Controller</code>	<code>SetFakeControllerContext</code>	Assigns a mock <code>HttpContextBase</code> and empty route data to the controller.
<pre>// Instantiate HomeController and assign it a fake HttpContext HomeController controller = new HomeController(); controller.SetFakeControllerContext();</pre>		
<code>HttpRequestBase</code>	<code>SetHttpRequestMethodResult</code>	Specifies the HTTP method of the request.
<pre>// Instantiate HomeController and assign it a fake HttpContext. The request is a POST HomeController controller = new HomeController(); controller.SetFakeControllerContext(); controller.SetHttpRequestMethodResult("POST");</pre>		
<code>HttpRequestBase</code>	<code>SetupRequestUrl</code>	Specifies the URL of the request.
<pre>// Instantiate HomeController and assign it a fake HttpContext. The request is a POST for http://www.mysite.com/Home/Index HomeController controller = new HomeController(); controller.SetFakeControllerContext(); controller.SetHttpRequestMethodResult("POST"); controller.Request.SetupRequestUrl ("http://www.mysite.com/Home/Index");</pre>		

In the following code samples, the `MvcMockHelpers` extension methods will be used to facilitate unit testing of ASP.NET MVC web applications.

Testing routes

When developing and using an ASP.NET MVC web application, ASP.NET routing plays an important role. The incoming URLs are mapped to a controller and an action method by the routing engine. Also, given a controller and action method name, the routing engine can map this into a real URL.

It may be useful to write unit tests for routes to ensure that URLs are mapped to the routes that you intended to map to. This may also prove to be useful when creating new route maps – if one existing route is broken, you'll notice that the corresponding unit tests fail.

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (`MvcTestingRoutesExample`). Let's test the following routes:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

routes.MapRoute(
    "ArchiveRoute",
    "Archive/{year}/{month}/{day}/{title}.htm",
    new { controller = "Home", action = "About" },
    new { year = @"\d{4}", month = @"\d{2}", day = @"\d{2}" }
);

routes.MapRoute(
    "Default",           // Route name
    "{controller}/{action}/{id}", // URL with parameters
    new { controller = "Home", action = "Index", id = "" } //
    Parameter defaults
);
```

In the above code snippet, three routes are registered. The first route ignores requests for any HTTP handler (named `*.axd`). The second route allows an archive to be retrieved, by using URLs such as `http://localhost:51741/Archive/2008/10/10/some-title.htm`. The last route is the default route, which catches any request that can be mapped into a controller and action.

The above route `ArchiveRoute` can be tested using the `MvcMockHelpers` extension methods:

```
[TestMethod]
public void ArchiveRouteTest()
{
    // Register routes
    RouteCollection routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    // Create a fake request
    HttpContextBase httpContext = MvcMockHelpers.FakeHttpContext(
        "~/Archive/2008/10/02/some-title.htm");

    // Retrieve route
    RouteData target = routes.GetRouteData(httpContext);

    // Verify
    Assert.AreEqual("2008", target.Values["year"]);
    Assert.AreEqual("10", target.Values["month"]);
    Assert.AreEqual("02", target.Values["day"]);
    Assert.AreEqual("some-title", target.Values["title"]);
    Assert.AreEqual("Home", target.Values["controller"]);
    Assert.AreEqual("About", target.Values["action"]);
}
```

In the above code sample, an empty route table is created and populated using your ASP.NET MVC application's `RegisterRoutes` method. After this, a mocked `HttpContext` is created for the URL `~/Archive/2008/10/02/some-title.htm`.

The current route can be determined by querying the `GetRouteData` method of `RouteCollection`, passing in the mocked `HttpContext` as a parameter. The routing engine is expected to return each component separately: year, month, day, title, controller, and action. All of these route values should be present and correct in order for the test to succeed.

Testing UpdateModel scenarios

Instead of mocking form post variables, the ASP.NET MVC framework provides the possibility to pass in form variables as an action method parameter when the `UpdateModel` or `TryUpdateModel` method is used. The `UpdateModel` (or `TryUpdateModel`) can update an existing object with variables posted in the HTTP request using `ModelBinders` (see elsewhere in this book).

The example in this topic is based on an ASP.NET MVC web application, which can be found in the sample code for this book (MvcUpdateModelExample).

Have a look at the following action method:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult New(FormCollection form)
{
    Employee employee = new Employee();
    try
    {
        UpdateModel(employee, new string[] { "Name", "Email" },
            form.ToValueProvider());

        if (string.IsNullOrEmpty(employee.Name)) throw new
            ArgumentNullException("Name");
        if (string.IsNullOrEmpty(employee.Email)) throw new
            ArgumentNullException("Email");

        return RedirectToAction("Index");
    }
    catch {
        return View("New", employee);
    }
}
```

The above action method is used to create a new `Employee`. It updates a new instance of an `Employee` with posted variables `Name` and `Email`. Whenever one of these is empty, an `Exception` is thrown and the view is re-rendered using the data that was previously entered. If everything is filled in, a redirect to the `Index` action method is performed.

Note that the `UpdateModel` method accepts a third parameter of type `IValueProvider`. The ASP.NET MVC `ModelBinder` infrastructure uses the `IValueProvider` interface to retrieve values for binding the model against. The `FormCollection` dictionary implements the interface `IValueProvider` and can, therefore, be used in conjunction with the `UpdateModel` and `TryUpdateModel` methods.

The following test creates a `FormCollection` that contains all of the required form values for binding to the `Employee` instance. This test asserts that a `RedirectToRouteResult` is returned, which indicates that the model has been updated successfully.

```
[TestMethod]
public void NewPostSucceeding()
{

```

```

// Arrange
FormCollection form = new FormCollection();
form.Add("Name", "Test");
form.Add("Email", "test@example.com");

HomeController controller = new HomeController();
controller.SetFakeControllerContext();
controller.Request.SetHttpMethodResult("POST");

// Act
RedirectToRouteResult result = controller.New(form) as
                                RedirectToRouteResult;

// Assert
Assert.IsNotNull(result);
}

```

A failing test can also be created. The following test creates a `FormCollection` that does not contain all of the required form values for binding to the `Employee` instance. This test asserts that a `ViewResult` is returned, and that its `ViewData.Model` contains an `Employee` instance.

```

[TestMethod]
public void NewPostFailing()
{
    // Arrange
    FormCollection form = new FormCollection();
    form.Add("Name", "Test");

    HomeController controller = new HomeController();
    controller.SetFakeControllerContext();
    controller.Request.SetHttpMethodResult("POST");

    // Act
    ViewResult result = controller.New(form) as ViewResult;

    // Assert
    ViewDataDictionary viewData = result.ViewData;
    Assert.IsInstanceOfType(viewData.Model, typeof(Employee));
}

```

Summary

In this chapter, we have learned some aspects of unit testing, and how to use them in an ASP.NET MVC web application. We've seen what unit testing is, and what its advantages are. We've also seen the different unit testing frameworks available, and have created our first unit test.

We've learned how we can generate unit tests, and how to use them in relation to testing action methods. We've later seen that a mocking framework can facilitate unit testing. Using a mocking framework, we've completed a unit test for the `Login` action method of the default `AccountController`.

Finally, we've used a mocking framework to mock ASP.NET components to test routes and model updates.

10

Hosting and Deployment

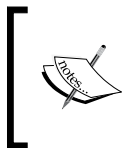
When building any application, the chances are that the application will have to be deployed. This chapter describes how you can deploy and host an ASP.NET MVC application in an Internet Information Server (IIS6 and IIS7) platform.

You will learn the following in this chapter:

- Which hosting platforms can be used to host an ASP.NET MVC web application
- The differences between IIS 7.0 integrated mode and classic mode
- How to create a wildcard script map in IIS 7.0 and IIS 6.0
- How to modify the route table to support ASP.NET routing in some hosting environments

Platforms that can be used

Theoretically, any web server capable of running ASP.NET web applications should be capable of running an ASP.NET MVC web application. Supported platforms are Windows running any version of **Internet Information Services (IIS)**, from version 5.1 on.



Some people managed to get ASP.NET MVC web applications running on Mono, an open source implementation of the .NET framework, but this is not officially supported. More on this can be found on: <http://www.tobinharris.com/2008/4/3/asp-net-mvc-on-mono-osx>.

Building an ASP.NET MVC web application also means building URL routes. URL routing is a key part of the ASP.NET MVC framework and is, therefore, required to run on the IIS server. Depending on the version of IIS being used, additional configuration may be required in order to be able to take advantage of URL routing.

Any ASP.NET MVC web application will be able to run on the following versions of IIS:

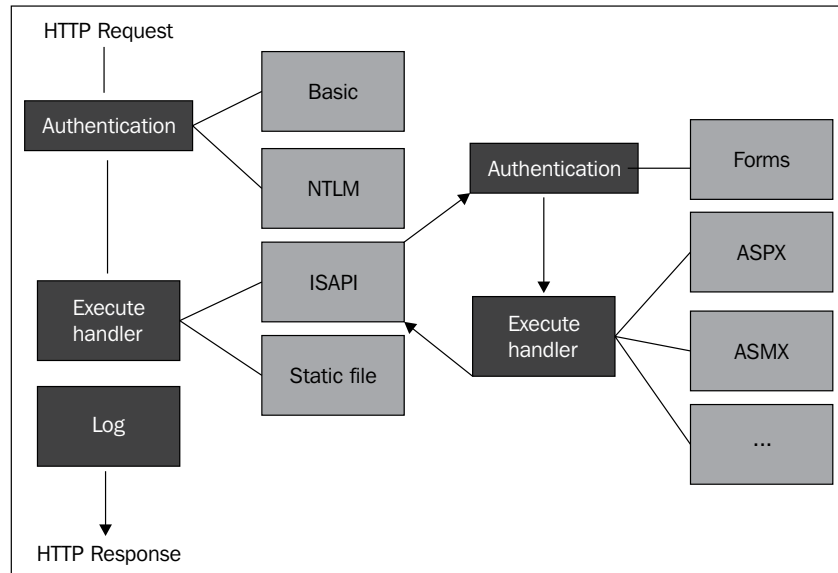
IIS version	Windows version	Remarks
IIS 7.0 (integrated mode)	Windows Server 2008	No special configuration required
	Windows Vista (except Home Basic)	
IIS 7.0 (classic mode)	Windows Server 2008	Special configuration required to use URL routing
	Windows Vista (except Home Basic)	
IIS 6.0	Windows Server 2003	Special configuration required to use URL routing
IIS 5.1	Windows XP Professional	Special configuration required to use URL routing
IIS 5.0	Windows 2000	Special configuration required to use URL routing

Differences between IIS 7.0 integrated and classic mode

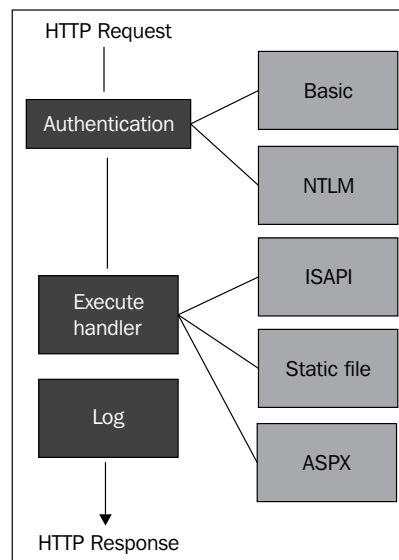
IIS 7.0 has been developed to be a flexible and scalable platform for hosting dynamic web applications including Microsoft ASP and ASP.NET.

When looking at ASP.NET, IIS 6.0 was built using ISAPI modules, requiring low-level C++ API calls and a lot of processing overhead when transferring an HTTP request to ASP.NET. For example, authentication was performed twice: once in IIS and once in ASP.NET. IIS 7.0. This introduced a whole new integrated model, which allowed ASP.NET applications to plug into the web server directly and actually become a part of the web server executable.

With classic mode, an HTTP request would be executed as follows:

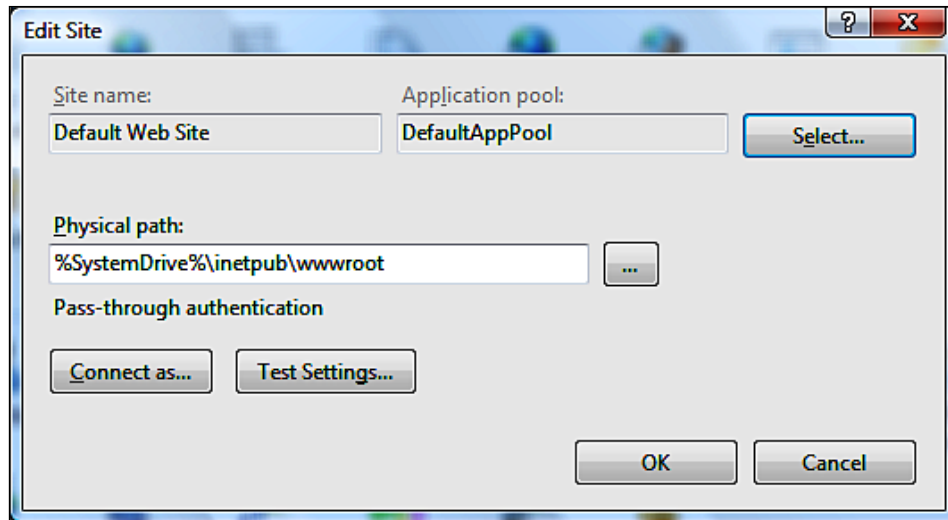


As you can see, things such as authentication are performed twice, only for ASP.NET requests. Protecting an image from being displayed using ASP.NET authentication would be impossible in the classic mode! Using integrated mode, any HTTP request can be processed using ASP.NET modules and handlers such as authentication, making ASP.NET a full member of the IIS request processing pipeline.



IIS 7.0 provides support both for this new integrated mode and for the classic IIS 6.0 mode. The first option allows you to configure IIS 7.0 from within your `web.config` (which is preconfigured for the ASP.NET MVC framework); the latter requires some server-side configuration. To check whether an application is running in integrated or classic mode, follow these steps:

1. Launch the **Internet Information Services Manager**.
2. In the **Connections** tree view, select an application.
3. In the **Actions** window, click on the **Basic Settings** link to open the **Edit Application** dialog box.
4. Verify the selected **Application pool**. If **DefaultAppPool** is selected, your application runs in an integrated mode and natively supports the ASP.NET MVC framework. If **Classic .NET AppPool** is selected, your application runs in the classic mode, and more configuration is required.



Hosting an ASP.NET MVC web application

If you or your web hosting provider have access to your web server's settings, a wildcard script map can be created in order to have full routing support. A wildcard script map enables you to map each incoming request into the ASP.NET framework. Be aware that this option passes every request into the ASP.NET framework (even images and CSS files!) and may have performance implications.

If you do not have access to the web server's settings, you can modify the route table to use file extensions. Instead of looking like this:

```
/Products/All
```

URLs would look like this:

```
/Products.aspx/All
```

This way, no configuration of the web server is required. It is, however, necessary to make some modifications to the application's route table.



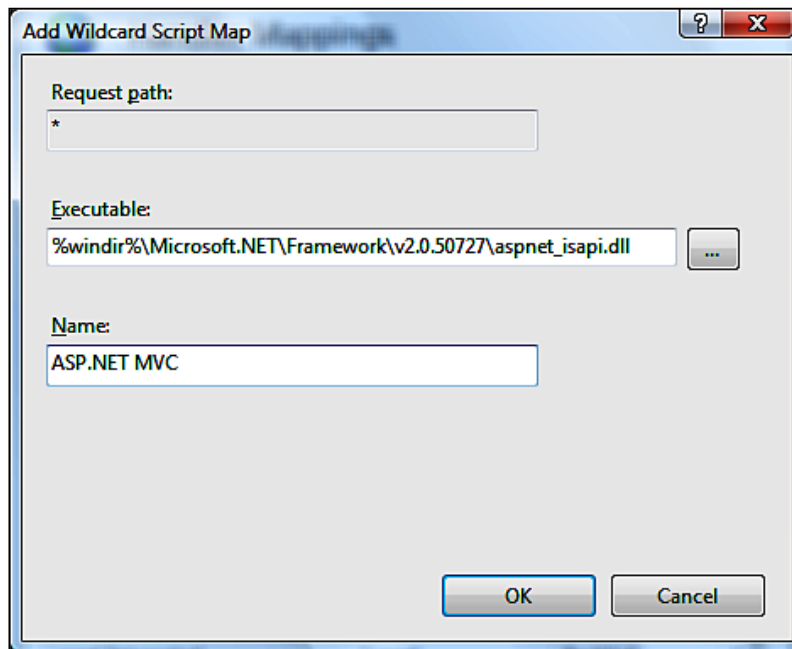
You do not have to configure anything if your IIS 7.0 server is operating in integrated mode.

Creating a wildcard script map in IIS 7.0

Here is how you can enable a wildcard script map in Internet Information Services 7.0:

1. Launch the **Internet Information Services Manager**.
2. In the **Connections** tree-view, select an application.
3. In the bottom toolbar, make sure that the **Features** view is selected.
4. Double-click on the **Handler Mappings** shortcut.
5. In the **Actions** window, click on the **Add Wildcard Script Map** button.
6. Enter the path to the `aspnet_isapi.dll` file, which is usually located in:
`%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_isapi.dll`.
7. Enter the name **ASP.NET MVC**.
8. Click on the **OK** button.

After doing this, any request for this specific web site will be executed by the ASP.NET engine.

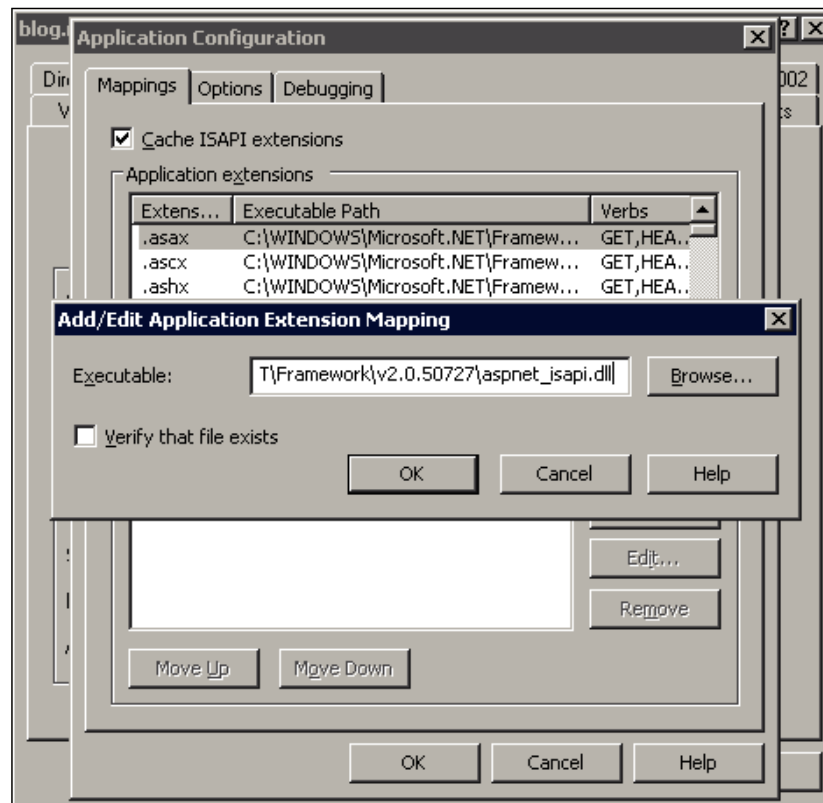


Creating a wildcard script map in IIS 6.0

Here is how you can enable a wildcard script map in Internet Information Services 6.0:

1. Launch the **Internet Information Services IIS Manager**.
2. Right-click on a web site and select **Properties**.
3. Select the **Home Directory** tab.
4. Near **Application** settings, click on the **Configuration** button.
5. Select the **Mappings** tab.
6. Near Wildcard application maps, click on the **Insert** button.
7. Enter the path to the `aspnet_isapi.dll` file, which is usually located in `%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_isapi.dll`
8. Uncheck the **Verify that file exists** checkbox.
9. Click on the **OK** button.

After following these steps, any request for this specific web site will be executed by the ASP.NET engine.



Modifying the route table to use file extensions

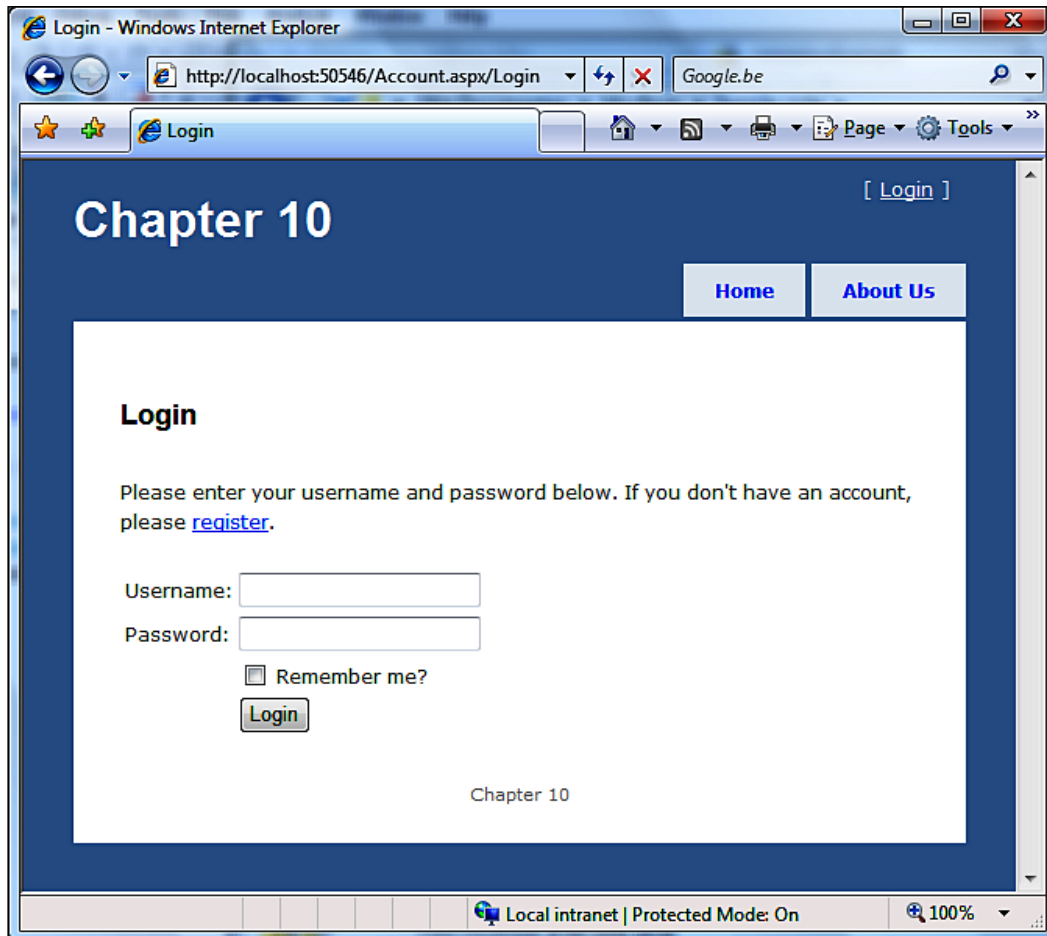
If you do not have access to your web server's settings and, therefore, cannot configure a wildcard script map, it is possible to modify the route table to use file extensions. Instead of looking like this:

```
/Products/All
```

URLs would look like this:

```
/Products.aspx/All
```

In the older versions of IIS, only certain requests are mapped to the ASP.NET framework. For example, only .aspx, .asmx, .ascx, and so on, are mapped to the ASP.NET framework. Extensions such as .htm, .jpg, .gif, and so on, are served directly by IIS without any ASP.NET processing being necessary. Because the .aspx extension is always mapped to the ASP.NET framework, it is an ideal candidate to trigger the routing engine.



In the Global.asax file of the web application, modify the default route to look like this:

```
using System.Web.Mvc;
using System.Web.Routing;
namespace ModifiedRouteExample
{
```

```

public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            "Default",
            "{controller}.aspx/{action}/{id}",
            new { controller = "Home", action = "Index", id = "" }
        );
    }
    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
}

```

The key difference between the standard route table and this modified route table is the .aspx extension:

```

routes.MapRoute(
    "Default",
    "{controller}.aspx/{action}/{id}",
    new { controller = "Home", action = "Index", id = "" }
);

```

All of the URLs in your application should now work with a pattern such as {controller}.aspx/{action}. If you are using hard-coded hyperlinks, make sure that you modify these links. Hyperlinks that are generated using the `ActionLink()` method of the `HtmlHelper` class should be updated automatically.

Summary

In this chapter, we have learned which hosting platforms can be used to host an ASP.NET MVC web application. We've also seen the differences between IIS 7.0 integrated mode and classic mode. We've learned how to create a wildcard script map in both IIS 7.0 and IIS 6.0. As an alternative to configuring the web server, we have learned how to modify the route table to support ASP.NET routing on some hosting environments.



Reference Application— CarTrackr

The sample application included with this book is an application I have written for various demos of the ASP.NET MVC framework. You can read its short description and download the latest version from www.codeplex.com/CarTrackr.

CarTrackr is an online software application designed to help you understand and track your fuel usage and kilometers driven.

You will have a record on when you filled up on fuel, how many kilometers you got in a given tank, how much you have spent, and how many liters of fuel you are using per 100 kilometers.

CarTrackr will enable you to improve your fuel economy and save money, as well as conserve fuel. Fuel economy and conservation is becoming an important way to control your finances in the current time of high prices.

Please note that this appendix will not cover the CarTrackr sample application in its entirety, but will zoom in on certain aspects that make developing ASP.NET MVC applications easier and faster.

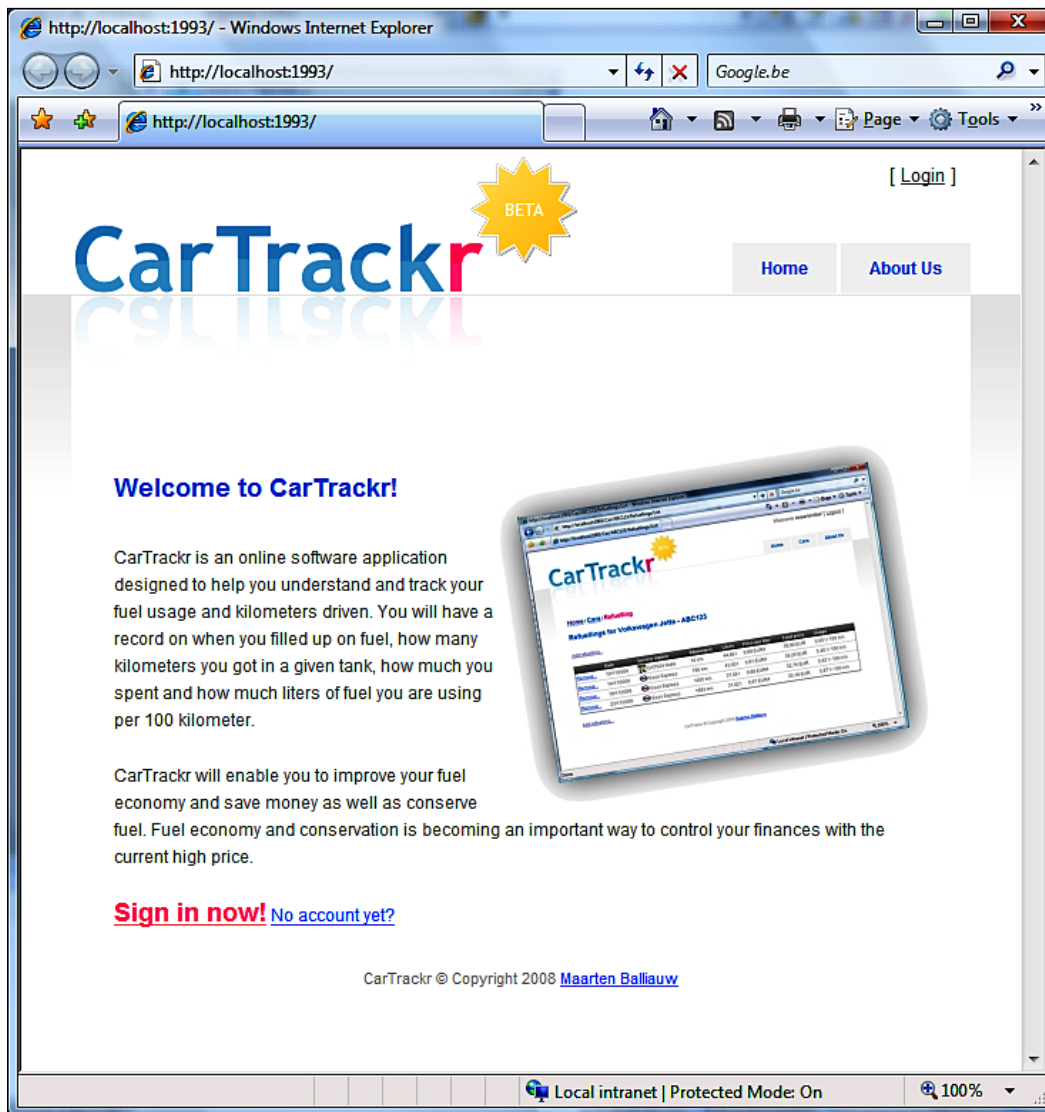
CarTrackr functionality

Before diving into the implementation details, the functionality of CarTrackr is explained by using the most important screens within the application.

Home page

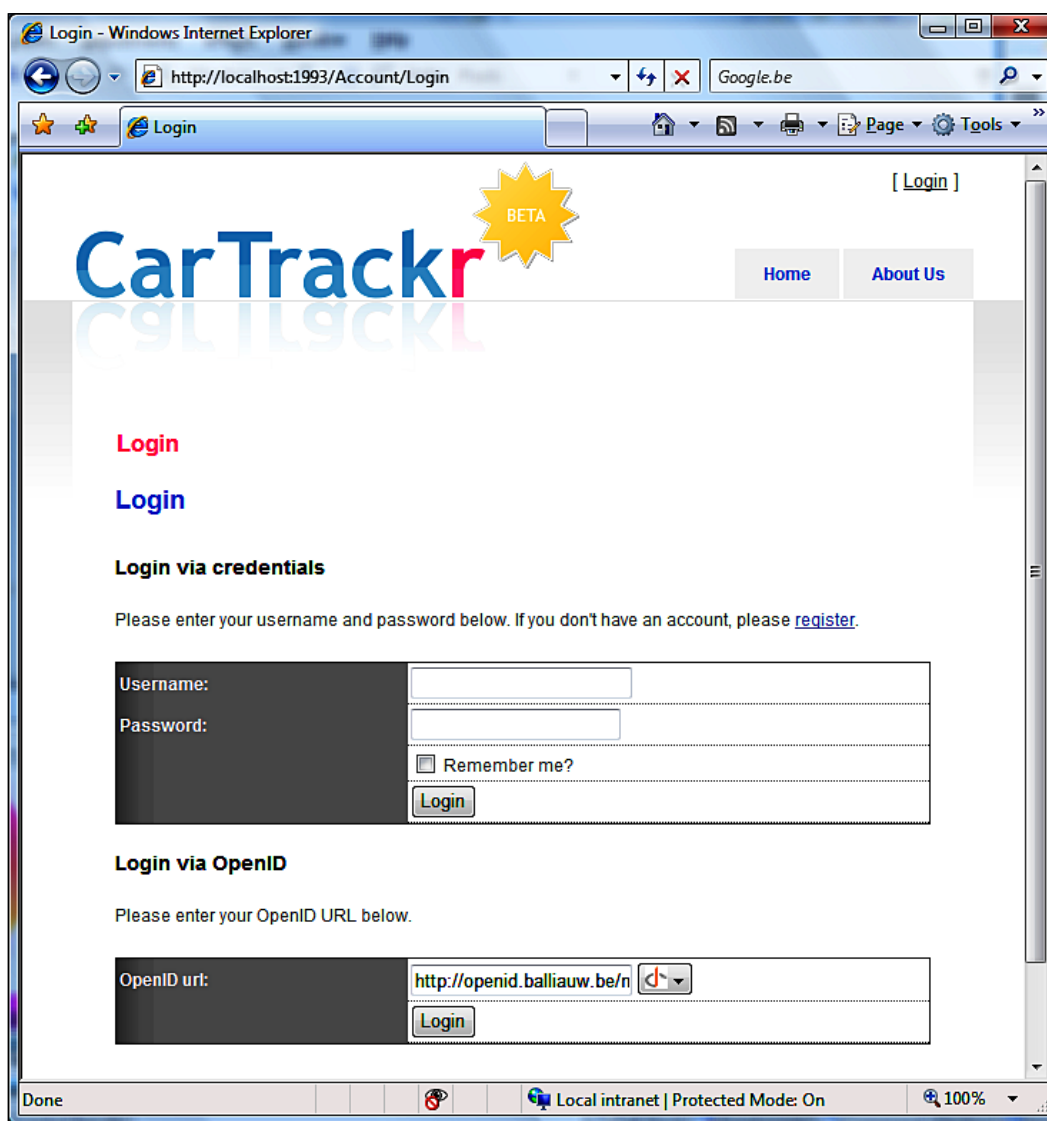
The CarTrackr home page looks quite simple. It gives a description of the application's purpose and features, and allows a user to sign in or register an account.

When a user has logged in, the home page will display a link that redirects the user to his or her list of tracked cars.



Login screen

In order to use the CarTrackr features, a user must be logged in. Logging in can be done by using the login screen and providing either a username or password combination, or an **OpenID** login URL.



The screenshot shows a web browser window titled "Login - Windows Internet Explorer". The address bar displays "http://localhost:1993/Account/Login". The page features the "CarTrackr" logo with a yellow "BETA" starburst. Navigation links for "Home" and "About Us" are present. The page is divided into two main sections: "Login via credentials" and "Login via OpenID".

Login via credentials

Please enter your username and password below. If you don't have an account, please [register](#).

Username:

Password:

☐ Remember me?

Login via OpenID

Please enter your OpenID URL below.

OpenID url:

Login via OpenID
Please enter your OpenID URL below.

OpenID url:

Sign in with OpenID using

Get an OpenID

myOpenID

AOL

Verisign

Technorati

Other OpenID

Flickr

Blogger

Vidooop

Vox

Yahoo!

Livejournal

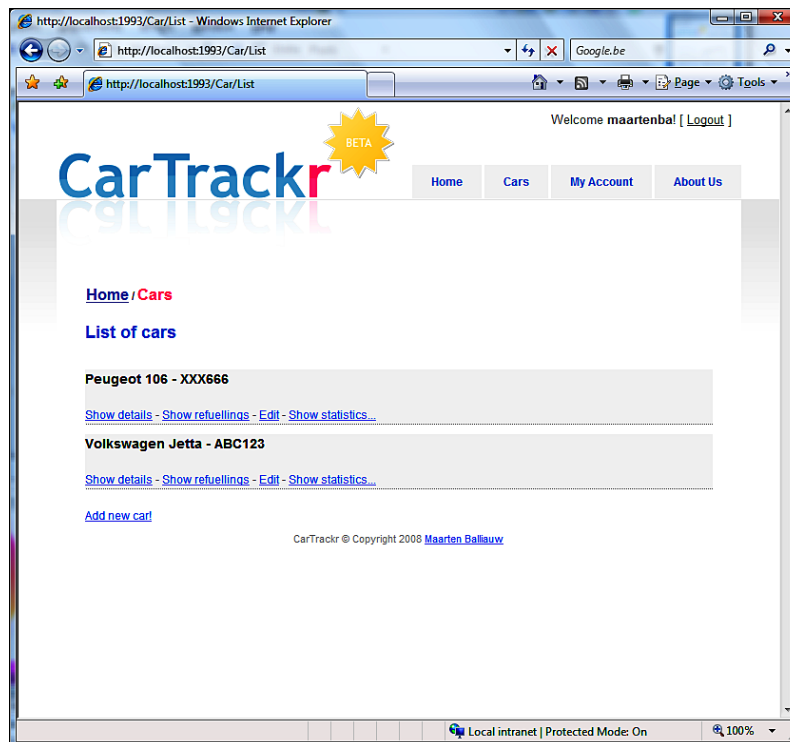
claimID

Wordpress

Help

List of cars

The main screen, from which any action in the CarTrackr application can be performed, is the list of cars. It displays a list of all tracked cars linked to the user's account, and allows the user to show more details for a car, show refuellings, and so on. There's one special link on this page: **Show statistics...**

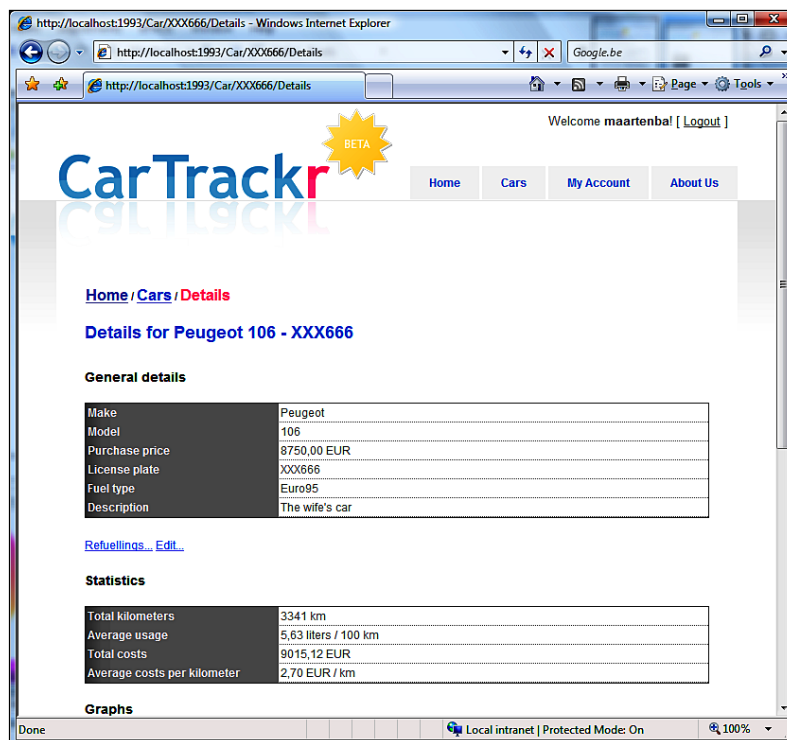


When a user clicks the **Show statistics...** link, a partial page update is performed by using an asynchronous web request (AJAX).

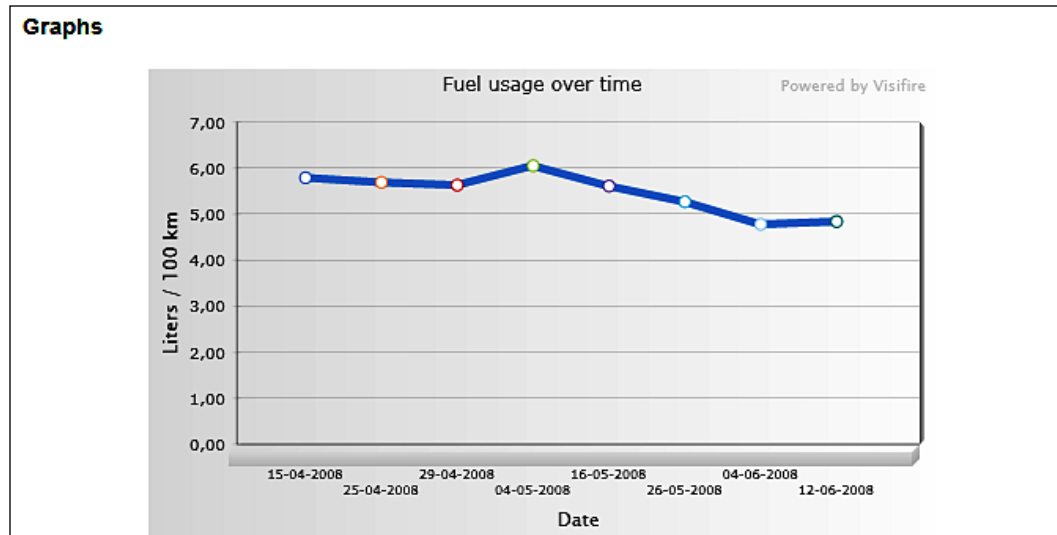
List of cars	
Peugeot 106 - XXX666	
Show details - Show refuellings - Edit - Show statistics...	
Total kilometers	3341 km
Average usage	5,63 liters / 100 km
Total costs	9015,12 EUR
Average costs per kilometer	2,70 EUR / km

Car details

Whenever a user requests a car's details, the details page is displayed. This page displays the properties for the car, for example, its **Make** and **Model**, along with some statistical data, for example, the **Average costs per kilometer** that the car consumes, the **Total kilometers**, and so on.



On the car details screen, the user can scroll down to see some graphs, which immediately show the car's statistics in a more visual manner. For example, the **Fuel usage over time** is plotted on a Silverlight graph:



Refuellings list

If required, the user can display a list of all refuellings for a car's lifetime. These refuellings are listed in a table. The user can remove a specific refuelling, or add a new refuelling. For each refuelling, key data is presented along with a small logo of the refuelling station brand.

http://localhost:1993/Car/XXX666/Refuellings/List - Windows Internet Explorer

http://localhost:1993/Car/XXX666/Refuellings/List

Google.be

http://localhost:1993/Car/XXX666/Refuellings/List

Welcome maartenba! [Logout]

CarTrackr BETA

Home Cars My Account About Us

[Home](#) / [Cars](#) / **Refuelling**

Refuellings for Peugeot 106 - XXX666

	Date	Service station	Kilometers	Liters	Price per liter	Total price	Usage
Remove...	04/04/2008	Texaco	15 km	37,19 l	1,12 EUR/l	41,76 EUR	0,00 l / 100 km
Remove...	15/04/2008	Esso Express	633 km	36,88 l	1,16 EUR/l	42,70 EUR	5,96 l / 100 km
Remove...	25/04/2008	Shell Express	1025 km	23,00 l	1,12 EUR/l	25,82 EUR	5,86 l / 100 km
Remove...	29/04/2008	Texaco	1266 km	14,00 l	1,13 EUR/l	15,81 EUR	5,80 l / 100 km
Remove...	04/05/2008	Shell Express	1700 km	27,00 l	1,10 EUR/l	29,70 EUR	6,22 l / 100 km
Remove...	16/05/2008	Q8	2011 km	18,00 l	1,21 EUR/l	21,78 EUR	5,78 l / 100 km
Remove...	26/05/2008	Shell Express	2544 km	29,00 l	1,23 EUR/l	35,67 EUR	5,44 l / 100 km
Remove...	04/06/2008	Q8	2896 km	17,45 l	1,31 EUR/l	22,89 EUR	4,95 l / 100 km
Remove...	12/06/2008	Q8	3341 km	22,32 l	1,30 EUR/l	28,99 EUR	5,01 l / 100 km

[Add refuelling...](#)

Local intranet | Protected Mode: On 100%

Data layer

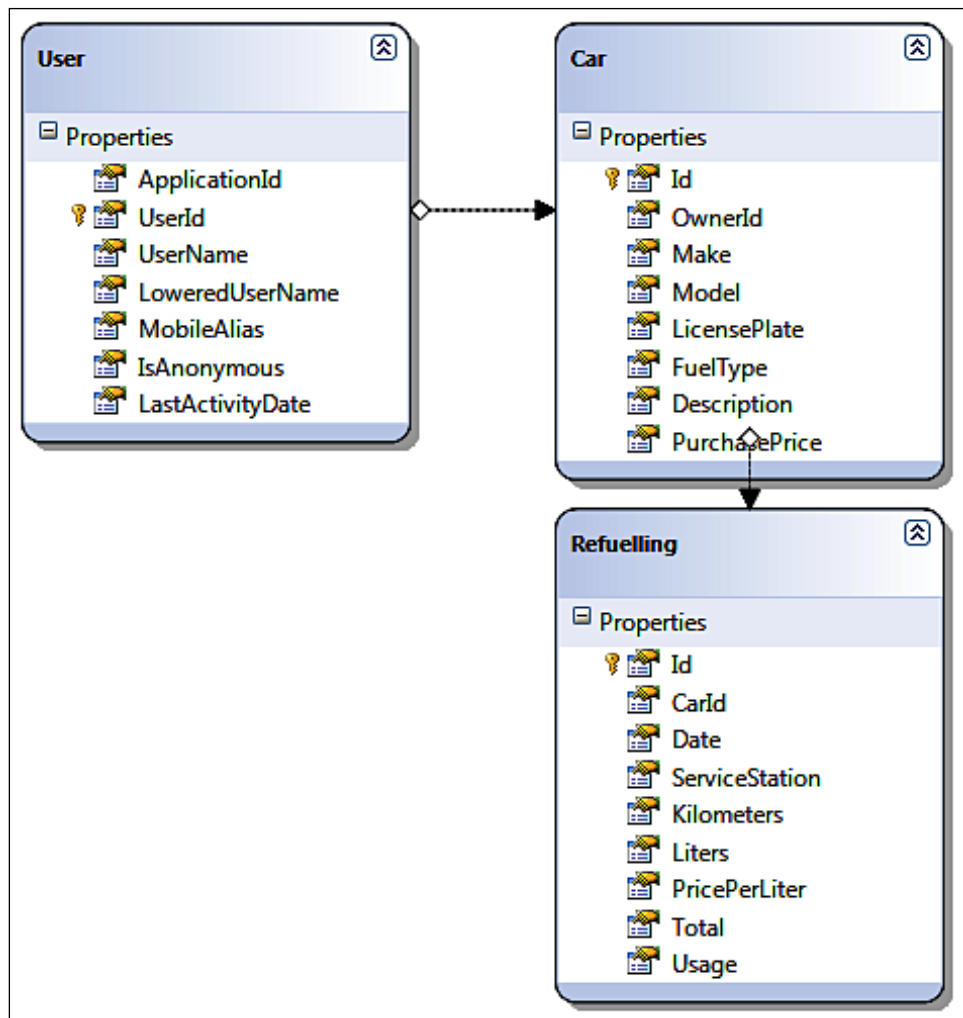
CarTrackr stores all of its data in a Microsoft SQL Server database, which is located in the **App_Data** folder of the CarTrackr Visual Studio project. This topic will describe how data is retrieved from the database in a model-driven, maintainable manner.

Linq to SQL model

As the underlying data source, CarTrackr uses a SQL server database. This database consists of the standard ASP.NET tables (aspnet_*) and two CarTrackr-specific tables: **Car** and **Refuelling**.

The database behind CarTrackr is exposed in the application using a LINQ to SQL model, which maps three domain classes to the database:

1. User: Mapping to ASP.NET's user table.
2. Car: Mapping to CarTrackr's car table.
3. Refuelling: Mapping to the refuelling table.



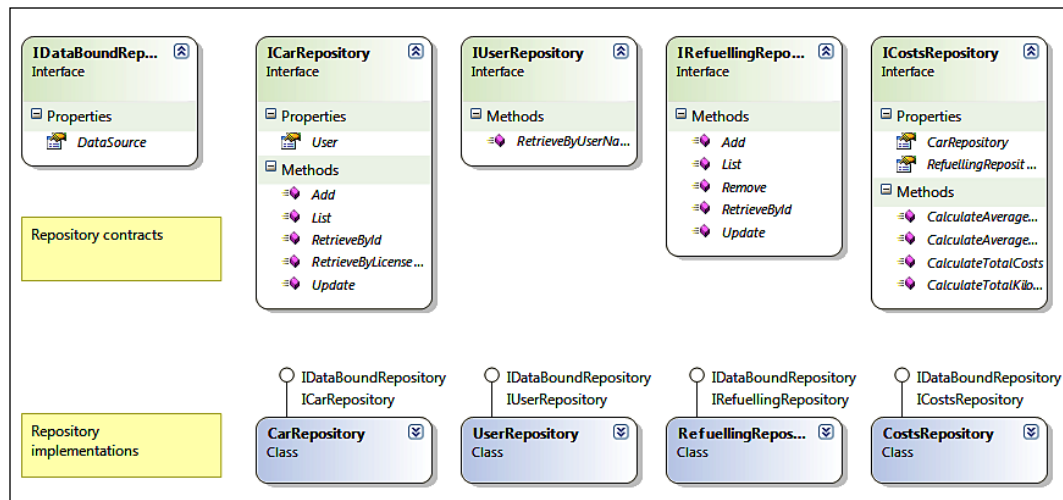
In the previous image, you can see which domain classes are used in the CarTrackr application. All functionality is mapped to these classes, which are in turn mapped to the SQL database using LINQ to SQL.

Repository pattern

According to Martin Fowler, the repository pattern provides a layer of abstraction over the mapping layer, with query construction code being concentrated to minimize duplicate query logic. In practice, the repository pattern is usually a collection of data access services, grouped in a way similar to the domain model classes. For example, `CarRepository` will provide all data access logic for a `Car`.

By accessing repositories via interfaces, the repository pattern helps to break the dependency between the domain model and data access code.

To facilitate application development, CarTrackr is built using the repository pattern. In short, every source of data is described using an interface. This interface is then used to communicate with the data layer. As an example, the `ICarRepository` interface exposes `Add`, `List`, `RetrieveById`, and other such methods. The `CarController` makes use of this `ICarRepository` interface's implementation, `CarRepository`. The `CarRepository` implementation itself uses the LINQ to SQL model to communicate with the database.



By using the repository pattern, CarTrackr does not have to know that the underlying data communications are done using LINQ to SQL. Instead, it only knows the methods used to retrieve and store data. One of the advantages of this approach is that unit tests can be written in a dummy (or mocked) repository without altering the CarTrackr application. This means that unit tests can be run on CarTrackr without requiring an actual database!

Dependency injection

Dependency injection is the process of making software components more loosely coupled by injecting dependencies into objects rather than having the object create its own dependencies.

As an example, take the following code:

```
public class Car
{
    private Engine engine;
    public Car()
    {
        engine = new Engine();
    }
}
```

In this example, the Car creates its own dependencies, in this case, a new Engine. It would be a difficult task to unit-test this Car class with a mocked Engine, because there is actually no way to control the dependency of the Car on the Engine. Also, when maintaining this application in the future, inserting other Engine types might be a lot of work and require the refactoring of code.

To solve this issue, the Car class can be rewritten as follows:

```
public class Car
{
    private Engine engine;
    public Car(Engine engineToUse)
    {
        engine = engineToUse;
    }
}
```

The application using `Car` can now create new `Car` instances and pass in any `Engine` instance—even future engines:

```
Car someCar = new Car(new Engine());
Car fastCar = new Car(new LightSpeedEngine());
```

Using dependency injection in applications allows you to prepare for future changes without requiring a lot of re-work. It also facilitates unit testing using mock objects.

How CarTrackr controllers are built

`CarTrackr` controllers are built using dependency injection. All repository implementations are passed into the controllers using parameters in the constructor.

```
public class RefuellingController : Controller
{
    private IUserRepository UserRepository;
    private ICarRepository CarRepository;
    private IRefuellingRepository RefuellingRepository;

    public RefuellingController(IUserRepository userRepository,
        ICarRepository carRepository, IRefuellingRepository
        refuellingRepository)
    {
        UserRepository = userRepository;
        CarRepository = carRepository;
        RefuellingRepository = refuellingRepository;
    }

    // ... action methods ...

    public ActionResult List(string licensePlate)
    {
        Car car = CarRepository.RetrieveByLicensePlate(licensePlate);
        List<Refuelling> refuellings = RefuellingRepository.List(car);
        var viewData = new RefuellingListViewData
        {
            Car = car,
            Refuellings = refuellings
        };
        return View("List", viewData);
    }
}
```

In the above code sample for the `RefuellingController`, three repository contracts are used: `IUserRepository`, `ICarRepository`, and `IRefuellingRepository`.

Concrete implementations of these dependencies can be passed into the `RefuellingController` by passing them as parameters in the constructor. Note that there is no parameter-less constructor, which forces developers to actually specify dependencies for this controller using its constructor.

The `List` action method, for example, uses the `CarRepository` and the `RefuellingRepository` dependencies to retrieve concrete data, which is passed into the view.

Using Unity for dependency injection

Unfortunately, there is one downside to using dependency injection in ASP.NET MVC controllers. How does the ASP.NET MVC framework know which dependencies should be passed into the constructor?

This can be achieved by using a dependency injection container, combined with some plumbing code. Dependency injection is the process of avoiding hard-coded dependencies in classes by passing dependencies in the constructor, as demonstrated in the previous topic. There are quite a few dependency injection containers for the .NET platform: Windsor, NInject, Unity, and so on. These containers will automatically map specific dependencies within the classes in your code, thereby assisting in creating loosely coupled applications.

CarTrackr uses the Unity application block as a dependency injection container. Unity is written by Microsoft's patterns & practices team and can be found on <http://www.codeplex.com/unity>. The following code can be found in CarTrackr's `Global.asax.cs`:

```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        // ...
    }

    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
        RegisterDependencies();
    }

    protected static void RegisterDependencies() {
        IUnityContainer container = new UnityContainer();

        // Registrations
        container.RegisterType<IFormsAuthentication,
            FormsAuthenticationWrapper>();
    }
}
```

```

        container.RegisterType<MembershipProvider>(Membership.
Provider);

        container.RegisterType<CarTrackrData, CarTrackrData>(
            new ContextLifetimeManager<CarTrackrData>());
        container.RegisterType<ICarRepository, CarRepository>(
            new ContextLifetimeManager<ICarRepository>());
        container.RegisterType<IUserRepository, UserRepository>(
            new ContextLifetimeManager<IUserRepository>());
        container.RegisterType<IRefuellingRepository,
            RefuellingRepository>(
            new ContextLifetimeManager<IRefuellingRepository>());
        container.RegisterType<ICostsRepository, CostsRepository>(
            new ContextLifetimeManager<ICostsRepository>());

        // Set controller factory
        ControllerBuilder.Current.SetControllerFactory(
            new UnityControllerFactory(container)
        );
    }
}

```

Whenever the CarTrackr application is started, some actions are executed. First of all, as with any normal ASP.NET MVC application, the route table is registered. Next, an *IUnityContainer* is instantiated. This *IUnityContainer* instance will be responsible for injecting class instances into constructors. After that, a lot of types are registered on this *IUnityContainer*, for example:

```

        container.RegisterType<ICarRepository, CarRepository>(
            new ContextLifetimeManager<ICarRepository>());

```

The above method call instructs the *IUnityContainer* to create a new *CarRepository* instance whenever a constructor requires an *ICarRepository*. Also, the lifetime of this *CarRepository* instance will be one request/response cycle to ensure that each request has its own *CarRepository* instance.

Basically, all repository implementations are registered with the Unity container in the same manner. After all registrations have been done, the following code snippet is executed:

```

// Set controller factory
ControllerBuilder.Current.SetControllerFactory(
    new UnityControllerFactory(container)
);

```

Remember the fact that the ASP.NET MVC framework allows you to plug in customized components at any point in its request life cycle. This has been explained in Chapter 4, *Components in the ASP.NET MVC Framework*. In this case, instantiating a controller is delegated to a custom class, *UnityControllerFactory*, instead of ASP.NET's default controller builder. The custom *UnityControllerFactory* overrides ASP.NET MVC's *DefaultControllerFactory*:

```
public class UnityControllerFactory : DefaultControllerFactory
{
    IUnityContainer container;

    public UnityControllerFactory(IUnityContainer container)
    {
        this.container = container;
    }

    protected override IController GetControllerInstance(Type
        controllerType)
    {
        IController controller = null;
        if (controllerType != null)
        {
            if (!typeof(IController).IsAssignableFrom(controllerType))
                throw new ArgumentException(string.Format(
                    "Type requested is not a controller: {0}",
                    controllerType.Name),
                    "controllerType");
            controller = container.Resolve(controllerType) as
                IController;
        }
        return controller;
    }
}
```

The *UnityControllerFactory* requires an *IUnityContainer* instance in its constructor, and will use this to instantiate a controller when required. The *GetControllerInstance* override instructs the *IUnityContainer* to check whether registrations for the controller type have been made. In the *CarTrackr* case, this method will inject all required repositories into each controller.

ASP.NET MVC Membership Starter Kit

The ASP.NET MVC Membership Starter Kit project provides a base web site and class library, which extends the default ASP.NET MVC framework project template with user and role administration, OpenID authentication, Windows Live authentication, and so on. It can be found at <http://www.codeplex.com/MvcMembership>.

CarTrackr uses the ASP.NET MVC Membership Starter Kit to provide OpenID authentication on the login page.

Login via OpenID

Please enter your OpenID URL below.

OpenID uri:

Sign in with OpenID using [Get an OpenID](#)

[myOpenID](#)
[Flickr](#)
[Yahoo!](#)

[AOL](#)
[Blogger](#)
[Livejournal](#)

[Verisign](#)
[Vidoop](#)
[claimID](#)

[Technorati](#)
[Vox](#)
[Wordpress](#)

[Other OpenID](#)
[Help](#)

Using the OpenID features from the ASP.NET MVC Membership Starter Kit means adding three methods to the AccountController: `XRDS`, `OpenIdLogin`, and `AssociateOpenIdIdentityToUserName`.

The `XRDS` action method simply returns a view that tells any OpenID provider where all of the OpenID-protected pages are located in the CarTrackr. The `OpenIdLogin` action method determines the login status of an OpenID provider and manages all login functionality.

Finally, the `AssociateOpenIdIdentityToUsername` is a method that is a custom CarTrackr implementation that associates an OpenID user with an ASP.NET user.

```
protected string AssociateOpenIdIdentityToUserName(string
    openIdIdentity)
{
    // Try to get user
    MembershipUser user = Provider.GetUser(openIdIdentity, true);
    // If we didn't find user, create a new one
    if (user == null)
    {
        string password = Guid.NewGuid().ToString();
        MembershipCreateStatus status;
        user = Membership.CreateUser(openIdIdentity, password,
            password + "@example.com", password, password, true,
            out status);
        if (status != MembershipCreateStatus.Success)
            throw new MembershipCreateUserException(status.ToString());
    }
    return (user == null ? null : user.UserName);
}
```

The `AssociateOpenIdIdentityToUsername` method checks to see if an OpenID identity can be linked to an existing user in the ASP.NET membership database. If not, a new user is created based on the OpenID identity. By doing this, cars and refuellings can be linked to an ASP.NET user, which represents an OpenID identity.

Form validation

CarTrackr contains various scenarios where form validation is performed. Let's have a look at one of these scenarios – creating a new Car. The action method `New` is defined in the `CarController`:

```
[AcceptVerbs("POST")]
[ValidateAntiForgeryToken]
public ActionResult New(FormCollection form)
{
    Car car = new Car();
    try
    {
        this.UpdateModel(car, new[] { "Make", "Model",
            "PurchasePrice", "LicensePlate", "FuelType",
            "Description" });
        CarRepository.Add(car);
    }
```

```

        return RedirectToAction("Details", new { licensePlate =
                                   car.LicensePlate });
    }
    catch (RuleViolationException)
    {
        this.UpdateModelStateWithViolations(car, ViewData.ModelState);
        return View("New", car);
    }
}

```

When creating a new Car, the New action method of the CarController class creates a new Car instance and tries to update this object with the data received from the posted form, by using the UpdateModel method. Immediately after that, the CarRepository is instructed to add the car instance and save it to the database. Note that this method will call the EnsureValid method that is defined on the Car class. The EnsureValid method checks whether there are any violations in the car instance, and throws a RuleViolationException if any errors are present.

```

public void EnsureValid()
{
    List<RuleViolation> issues = GetRuleViolations();
    if (issues.Count != 0)
        throw new RuleViolationException("Business Rule Violations",
                                           issues);
}

```

The GetRuleViolations method performs various checks on the car instance. Whenever a check fails, a new RuleViolation is added to a list of violations. The RuleViolation object holds the property name, the provided value and an error message.

```

public List<RuleViolation> GetRuleViolations()
{
    List<RuleViolation> validationIssues = new List<RuleViolation>();
    if (string.IsNullOrEmpty(Make))
        validationIssues.Add(new RuleViolation("Make", Make, "Make
        should be specified!"));
    if (string.IsNullOrEmpty(Model))
        validationIssues.Add(new RuleViolation("Model", Model, "Model
        should be specified!"));
    if (PurchasePrice <= 0)
        validationIssues.Add(new RuleViolation("PurchasePrice",
        PurchasePrice, "Purchase price should be specified!"));
    if (string.IsNullOrEmpty(LicensePlate))

```

```
        validationIssues.Add(new RuleViolation("LicensePlate",  
            LicensePlate, "License plate should be specified!"));  
    return validationIssues;  
}
```

When the New action method of CarController catches a `RuleViolationException`, it calls the `UpdateModelStateWithViolations` method (and provides the car being added) and the `ViewData ModelState` dictionary. The `UpdateModelStateWithViolations` method copies all of the rule violations from the `RuleViolationException` into the `ViewData ModelState` dictionary. Afterwards, the CarController renders the view again, which will now display any validation issues:

Home / Cars / New

Add new car

Make		Make should be specified!
Model		Model should be specified!
Purchase price	0.00	Purchase price should be specified!
License plate	List	
Fuel type		
Description		

Save Cancel

Error messages in the view are displayed using ASP.NET MVC's `HtmlHelper.ValidationMessage` form helper, which we explained in Chapter 4, *Components in the ASP.NET MVC Framework*.

ASP.NET provider model

The ASP.NET MVC framework uses the ASP.NET provider model. This enables developers to re-use ASP.NET code in ASP.NET MVC applications.

Currently, the ASP.NET MVC framework does not support having a sitemap base on controller names and action method names. In August 2008, I created a huge blog post on implementing a custom sitemap provider for the ASP.NET MVC framework. You can read the blog post on <http://tinyurl.com/4pr557>.

Using the sitemap provider from my blog post allows you to create a sitemap file based on controller names and action method names:

```
<?xml version="1.0" encoding="utf-8" ?>
<sitemap>
  <siteMapNode id="Root" url="~/Home/Index">
    <mvcSiteMapNode id="About" title="About Us" controller="Home"
action="About" />
    <mvcSiteMapNode id="Cars" title="Cars" controller="Car"
action="Index">
      <mvcSiteMapNode id="DetailsCar" title="Details" controller="Car"
action="Details" />
      <mvcSiteMapNode id="EditCar" title="Edit" controller="Car"
action="Edit" />
      <mvcSiteMapNode id="NewCar" title="New" controller="Car"
action="New" />
      <mvcSiteMapNode id="Refuelling" title="Refuelling"
controller="Refuelling" action="List" licensePlate="">
        <mvcSiteMapNode id="NewRefuelling" title="New"
controller="Refuelling" action="New" />
      </mvcSiteMapNode>
    </mvcSiteMapNode>
    <mvcSiteMapNode id="Account" title="Account" controller="Account"
action="Index">
      <mvcSiteMapNode id="Login" title="Login" controller="Account"
action="Login" />
      <mvcSiteMapNode id="Register" title="Account Creation"
controller="Account" action="Register" />
      <mvcSiteMapNode id="ChangePassword" title="Change Password"
controller="Account" action="ChangePassword" />
      <mvcSiteMapNode id="Logout" title="Logout" controller="Account"
action="Logout" />
    </mvcSiteMapNode>
  </siteMapNode>
</sitemap>
```

Each sitemap node lets you specify an ID for the node (required), a title that will be used in the ASP.NET sitemap controls, a controller name, an action name, and other optional parameters. For example, a sitemap node for the change password action on the AccountController would look like this:

```
<mvcSiteMapNode id="ChangePassword" title="Change Password"
controller="Account" action="ChangePassword" />
```

In order to let ASP.NET standard controls pick up a reference to this sitemap, the following code has been added to the `system.web` section of the application's `Web.config` file:

```
<siteMap defaultProvider="MvcSitemapProvider">
  <providers>
    <add name="MvcSitemapProvider"
        type="CarTrackr.Core.MvcSitemapProvider"
        siteMapFile="~/Web.sitemap"
        securityTrimmingEnabled="true"
        cacheDuration="1"/>
  </providers>
</siteMap>
```

The above code registers the custom sitemap provider implementation as the default sitemap provider for the CarTrackr application. It also enables `securityTrimming`, which instructs the sitemap provider to show or hide certain sitemap nodes based on security aspects.

[Home](#) / [Cars](#) / [New](#)

After configuring the sitemap provider, the standard ASP.NET sitemap controls can be used. For example, the `SiteMapPath` control is used in the CarTrackr's master page:

```
<asp:SiteMapPath ID="breadCrumbTrail" runat="server">
  <NodeStyle ForeColor="#0063DC" Font-Bold="true" Font-Size="1.4em" />
  <CurrentNodeStyle ForeColor="#FF0084" />
  <PathSeparatorStyle ForeColor="#000000" Font-Bold="true" />
  <PathSeparatorTemplate> / </PathSeparatorTemplate>
  <RootNodeTemplate><a href="~/
    runat="server">Home</a></RootNodeTemplate>
</asp:SiteMapPath>
```

Unit testing CarTrackr

The CarTrackr application has been designed with testability in mind. Each component can be mocked by using a mocking framework, for example Moq. More on mocking frameworks can be found in Chapter 9, *Testing an Application*.

A real database server is not needed for running tests—it would slow down the process of running tests and could possibly fail the tests if the connection is suddenly dropped. Instead, the repository design pattern is used to abstract data access implementation, allowing a custom, dummy data layer to be used in unit testing.

Unit tests in CarTrackr

All unit tests in CarTrackr are written using the `MvcMockHelpers` extension methods, which can be found in Appendix B. This allows unit tests to create a fake web server environment without the need for a real IIS instance to run the tests on. In each test, the Moq mocking framework is used to create a mocked implementation of classes that are used by the controller.

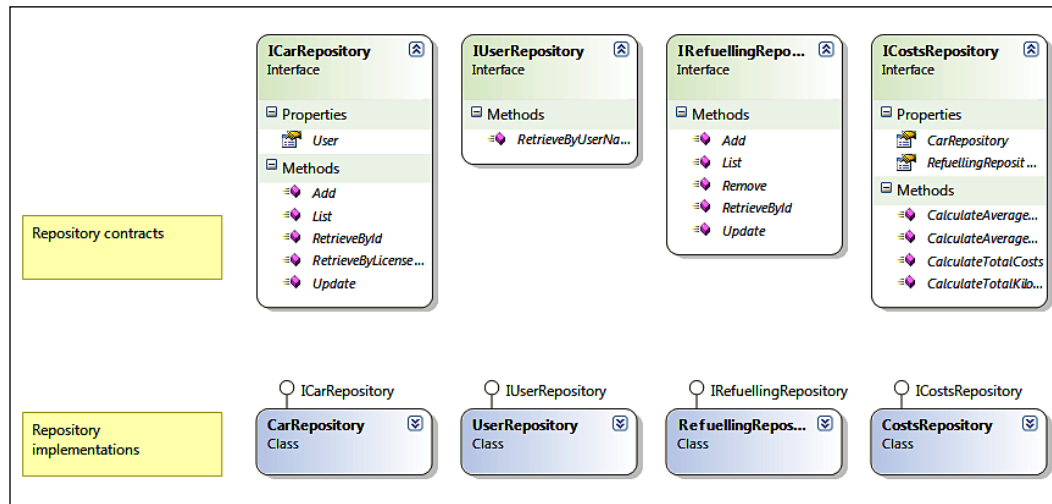
```
/// <summary>
///A test for Logout
///</summary>
[TestMethod()]
public void LogoutTest()
{
    // Setup
    var formsAuthenticationMock = new Mock<IFormsAuthentication>();
    var membershipProviderMock = new Mock<MembershipProvider>();
    formsAuthenticationMock.Expect(f => f.SignOut()).AtMostOnce();
    AccountController target = new AccountController
        (formsAuthenticationMock.Object, membershipProviderMock.Object);
    target.SetFakeControllerContext();

    // Execute
    RedirectToRouteResult result = target.Logout() as
        RedirectToRouteResult;
}
```

In the above test for the `Logout` action method of `AccountController`, a mocked `IFormsAuthentication` class is used by the `AccountController`. This mock is instructed to expect a call to its `SignOut` method at least once. If this expectation fails, the unit test will also fail. In addition to this, the `SetFakeControllerContext` extension method of `MvcMockHelpers` is used to set up a fake web server environment.

Mock repository

Earlier in this appendix, the repository design pattern was explained. CarTrackr unit tests are using a custom repository implementation, based on the repository contracts that are defined in CarTrackr.



Each repository contract is implemented in the unit testing project, based on data from a **List**, rather than data from a database. This allows the set-up of different testing scenarios more easily, without the requirement to deploy a custom database to a real database server for each unit test. Another advantage of this approach is that unit testing becomes more flexible and unit testing speed is dramatically increased.

When unit testing a controller, for example, the **RefuellingController**'s **Remove** action method can be achieved by instantiating the controller with these custom repository implementations (setup part of the following unit test):

```
/// <summary>
///A test for Remove
///</summary>
[TestMethod()]
public void RemoveTest()
{
    // Setup
    DataStore dataStore = new DataStore();
    IUserRepository userRepository = new
        CarTrackr.Tests.Repository.UserRepository(dataStore);
    ICarRepository carRepository = new
        CarTrackr.Tests.Repository.CarRepository(dataStore);
```

```
IRefuellingRepository refuellingRepository = new
    CarTrackr.Tests.Repository.RefuellingRepository(dataStore);

RefuellingController target = new
    RefuellingController(userRepository, carRepository,
        refuellingRepository);

// Execute
Guid id = dataStore.Refuellings[0].Id;

RedirectToRouteResult result = target.Remove(id) as
    RedirectToRouteResult;

// Verify
Assert.IsNull(refuellingRepository.RetrieveById(id));
}
```

After the test has been run, the repository is queried for the deleted refuelling. The test expects the repository to return a null value rather than a `Refuelling` instance.

Summary

In this appendix, some tips and tricks for building a real-life ASP.NET MVC application were covered, using the CarTrackr sample application, by zooming in on certain aspects that make developing ASP.NET MVC applications easier and faster. Using LINQ to SQL in a flexible manner, combined with the repository design pattern and dependency injection, provides the application with a flexible manner for coping with future requirements, and facilitates easier unit testing using mock data sources.

The ASP.NET MVC framework uses the ASP.NET provider model: authentication, authorization, membership, and session data. Developers can re-use ASP.NET code based on the provider model in ASP.NET MVC applications. In the CarTrackr application, a custom sitemap provider is developed and used as the default sitemap provider for standard ASP.NET controls.

CarTrackr uses the ASP.NET MVC Membership Starter Kit project to provide OpenID authentication on the login page.

Server-side form validation scenarios were explained by using an `EnsureValid` method on the domain objects. This method provides a list of possible error messages, which can be mapped to the `ModelState` dictionary of `ViewData` and used in the view for displaying error messages.

B

ASP.NET MVC Mock Helpers

This appendix contains the source code that assists in testing an ASP.NET MVC application using a mocking framework, as described in Chapter 9, *Testing an Application*.

When unit testing ASP.NET MVC applications, ASP.NET components such as `Request` and `Response` are often mocked. These components are normally filled by the ASP.NET runtime, which is unavailable when performing unit tests. To make use of `HttpContext`, `Request`, `Response`, `SessionState`, and server variables, the `MvcMockHelpers` extension methods from this appendix can be used instead of firing up a web server to perform the tests.

A version of `MvcMockHelpers` is provided for three mocking frameworks: `RhinoMocks`, `Moq`, and `TypeMock`. An explanation of all of the methods can be found in Chapter 9.

The original source code can be found on Scott Hanselman's blog at <http://www.hanselman.com/blog/ASPNETMVCSessionAtMix08TDDAndMvcMockHelpers.aspx>.

RhinoMocks

Mocking framework URL: <http://ayende.com/projects/rhino-mocks.aspx>

```
using System;
using System.Web;
using Rhino.Mocks;
using System.Text.RegularExpressions;
using System.IO;
using System.Collections.Specialized;
using System.Web.Mvc;
using System.Web.Routing;

namespace UnitTests
{
```

```
public static class MvcMockHelpers
{
    public static HttpContextBase FakeHttpContext(this
        MockRepository mocks)
    {
        HttpContextBase context =
            mocks.PartialMock<HttpContextBase>();
        HttpRequestBase request =
            mocks.PartialMock<HttpRequestBase>();
        HttpResponseBase response =
            mocks.PartialMock<HttpResponseBase>();
        HttpSessionStateBase session =
            mocks.PartialMock<HttpSessionStateBase>();
        HttpServerUtilityBase server =
            mocks.PartialMock<HttpServerUtilityBase>();

        SetupResult.For(context.Request).Return(request);
        SetupResult.For(context.Response).Return(response);
        SetupResult.For(context.Session).Return(session);
        SetupResult.For(context.Server).Return(server);

        mocks.Replay(context);
        return context;
    }

    public static HttpContextBase FakeHttpContext(this
        MockRepository mocks, string url)
    {
        HttpContextBase context = FakeHttpContext(mocks);
        context.Request.SetupRequestUrl(url);
        return context;
    }

    public static void SetFakeControllerContext(this
        MockRepository mocks, Controller controller)
    {
        var httpContext = mocks.FakeHttpContext();
        ControllerContext context = new ControllerContext(new
            RequestContext(httpContext,
                new RouteData()), controller);
        controller.ControllerContext = context;
    }

    static string GetUrlFileName(string url)
    {
        if (url.Contains("?"))
            return url.Substring(0, url.IndexOf("?"));
        else
            return url;
    }
}
```

```

static NameValueCollection GetQueryStringParameters(string
    url)
{
    if (url.Contains("?"))
    {
        NameValueCollection parameters = new
            NameValueCollection();

        string[] parts = url.Split("?".ToCharArray());
        string[] keys = parts[1].Split("&".ToCharArray());
        foreach (string key in keys)
        {
            string[] part = key.Split("=".ToCharArray());
            parameters.Add(part[0], part[1]);
        }
        return parameters;
    }
    else
    {
        return null;
    }
}

public static void SetHttpMethodResult(this HttpRequestBase
    request, string httpMethod)
{
    SetupResult.For(request.HttpMethod).Return(httpMethod);
}

public static void SetupRequestUrl(this HttpRequestBase
    request, string url)
{
    if (url == null)
        throw new ArgumentNullException("url");
    if (!url.StartsWith("~/"))
        throw new ArgumentException("Sorry, we expect a
            virtual url starting with \"~/\".");
    SetupResult.For(request.QueryString).Return
        (GetQueryStringParameters(url));
    SetupResult.For(request.
        AppRelativeCurrentExecutionFilePath)
        .Return(GetUrlFileName(url));
    SetupResult.For(request.PathInfo).Return(string.Empty);
}
}
}

```

Moq

Mocking framework URL: <http://code.google.com/p/moq/>

Helpers ported by Kzu: <http://www.clariusconsulting.net/blogs/kzu/>

```
using System;
using System.Web;
using System.Text.RegularExpressions;
using System.IO;
using System.Collections.Specialized;
using System.Web.Mvc;
using System.Web.Routing;
using Moq;

namespace UnitTests
{
    public static class MvcMockHelpers
    {
        public static HttpContextBase FakeHttpContext()
        {
            var context = new Mock<HttpContextBase>();
            var request = new Mock<HttpRequestBase>();
            var response = new Mock<HttpResponseBase>();
            var session = new Mock<HttpSessionStateBase>();
            var server = new Mock<HttpServerUtilityBase>();

            context.Expect(ctx =>
                ctx.Request).Returns(request.Object);
            context.Expect(ctx =>
                ctx.Response).Returns(response.Object);
            context.Expect(ctx =>
                ctx.Session).Returns(session.Object);
            context.Expect(ctx =>
                ctx.Server).Returns(server.Object);

            return context.Object;
        }

        public static HttpContextBase FakeHttpContext(string url)
        {
            HttpContextBase context = FakeHttpContext();
            context.Request.SetupRequestUrl(url);
            return context;
        }

        public static void SetFakeControllerContext(this Controller
            controller)
        {
        }
    }
}
```

```
        var httpContext = FakeHttpContext();
        ControllerContext context = new ControllerContext(new
            RequestContext(httpContext,
                new RouteData()), controller);
        controller.ControllerContext = context;
    }

    static string GetUrlFileName(string url)
    {
        if (url.Contains("?"))
            return url.Substring(0, url.IndexOf("?"));
        else
            return url;
    }

    static NameValueCollection GetQueryStringParameters(string
        url)
    {
        if (url.Contains("?"))
        {
            NameValueCollection parameters = new
                NameValueCollection();

            string[] parts = url.Split("?".ToCharArray());
            string[] keys = parts[1].Split("&".ToCharArray());
            foreach (string key in keys)
            {
                string[] part = key.Split("=".ToCharArray());
                parameters.Add(part[0], part[1]);
            }

            return parameters;
        }
        else
        {
            return null;
        }
    }

    public static void SetHttpRequestResult(this HttpRequestBase
        request, string httpMethod)
    {
        Mock.Get(request)
            .Expect(req => req.HttpMethod)
            .Returns(httpMethod);
    }

    public static void SetupRequestUrl(this HttpRequestBase
        request, string url)
```

```
{
    if (url == null)
        throw new ArgumentNullException("url");
    if (!url.StartsWith("~/"))
        throw new ArgumentException("Sorry, we expect a
            virtual url starting with \"~/\".");
    var mock = Mock.Get(request);
    mock.Expect(req => req.QueryString)
        .Returns(GetQueryStringParameters(url));
    mock.Expect(req =>
        req.AppRelativeCurrentExecutionFilePath)
        .Returns(GetUrlFileName(url));
    mock.Expect(req => req.PathInfo)
        .Returns(string.Empty);
}
}
```

TypeMock

Mocking framework URL: <http://www.typemock.com>

Helpers ported by Roy Oshero: <http://www.iserializable.com>

```
using System;
using System.Collections.Specialized;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using TypeMock;

namespace Typemock.Mvc
{
    static class MvcMockHelpers
    {
        public static void SetFakeContextOn(Controller controller)
        {
            HttpContextBase context =
                MvcMockHelpers.FakeHttpContext();
            controller.ControllerContext = new ControllerContext(new
                RequestContext(context, new RouteData()), controller);
        }

        public static void SetHttpRequestResult(this HttpRequestBase
            request, string httpMethod)
```

```
{
    using (var r = new RecordExpectations())
    {
        r.ExpectAndReturn(request.HttpMethod, httpMethod);
    }
}

public static void SetupRequestUrl(this HttpRequestBase
    request, string url)
{
    if (url == null)
        throw new ArgumentNullException("url");
    if (!url.StartsWith("~/"))
        throw new ArgumentException("Sorry, we expect a
            virtual url starting with \"~/\".");
    var parameters = GetQueryStringParameters(url);
    var fileName = GetUrlFileName(url);
    using (var r = new RecordExpectations())
    {
        r.ExpectAndReturn(request.QueryString, parameters);
        r.ExpectAndReturn(request.
            AppRelativeCurrentExecutionFilePath, fileName);
        r.ExpectAndReturn(request.PathInfo, string.Empty);
    }
}

static string GetUrlFileName(string url)
{
    if (url.Contains("?"))
        return url.Substring(0, url.IndexOf("?"));
    else
        return url;
}

static NameValueCollection GetQueryStringParameters(string
    url)
{
    if (url.Contains("?"))
    {
        NameValueCollection parameters = new
            NameValueCollection();
        string[] parts = url.Split("?".ToCharArray());
        string[] keys = parts[1].Split("&".ToCharArray());
        foreach (string key in keys)
        {

```

```
        string[] part = key.Split("=").ToCharArray();
        parameters.Add(part[0], part[1]);
    }
    return parameters;
}
else
{
    return null;
}
}
public static HttpContextBase FakeHttpContext(string url)
{
    HttpContextBase context = FakeHttpContext();
    context.Request.SetupRequestUrl(url);
    return context;
}
public static HttpContextBase FakeHttpContext()
{
    HttpContextBase context =
        MockManager.MockObject<HttpContextBase>().Object;
    HttpRequestBase request =
        MockManager.MockObject<HttpRequestBase>().Object;
    HttpResponseBase response =
        MockManager.MockObject<HttpResponseBase>().Object;
    HttpSessionStateBase sessionState =
        MockManager.MockObject<HttpSessionStateBase>().Object;
    HttpServerUtilityBase serverUtility =
        MockManager.MockObject<HttpServerUtilityBase>().Object;
    using (var r = new RecordExpectations())
    {
        r.DefaultBehavior.RepeatAlways();
        r.ExpectAndReturn(context.Response, response);
        r.ExpectAndReturn(context.Request, request);
        r.ExpectAndReturn(context.Session, sessionState);
        r.ExpectAndReturn(context.Server, serverUtility);
    }
    return context;
}
}
```

C

Useful Links and Open Source Projects Providing Additional Features

This appendix contains URLs to web sites that provide information about the ASP.NET MVC framework, and open source projects that provide additional features.

Information portals

This section lists all sorts of useful portals related to the ASP.NET framework. These portals offer tutorials, videos, and learning resources, as well as news related to ASP.NET MVC.

ASP.NET/MVC

The ASP.NET MVC web site is the official web site related to the ASP.NET MVC framework. It offers a direct download link for the ASP.NET MVC framework installer, as well as lots of documentation, samples, and screencasts.

URL: <http://www.asp.net/mvc>

Useful Links and Open Source Projects Providing Additional Features

The screenshot shows the Microsoft ASP.NET MVC website. At the top, there's a navigation bar with links like Home, Get Started, Learn, Downloads, AJAX, Community, Wiki, and Forums. Below this, there's a section for ASP.NET MVC with a brief description and a list of benefits: Clear separation of concerns, Testability - support for Test-Driven Development, Fine-grained control over HTML and JavaScript, and Intuitive URLs. To the right, there's a featured advertisement for ASP.NET Web Hosting, offering 6 months free and no setup fees. Below the main content, there's a section for ASP.NET MVC Tutorials with icons for Overview, Routing, Controllers, Views, and Models. The bottom right corner features a section for upcoming ASP.NET MVC books.

Aspdotnetmvc.com

The ASP.NET MVC Information Portal offers an overview of all of the news, announcements, blog posts, tutorials, tips, and other resources regarding the ASP.NET MVC framework.

URL: <http://aspdotnetmvc.com>

The screenshot shows the ASPDotNetMVC website. At the top, there's a navigation bar with links like Home, Blogs, Books, Buzz, In Action, News, Resources, Videos, About, and Contact. Below this, there's a section for the ASP.NET MVC Framework (Model View Controller) with a brief description and a link to the ASP.NET MVC Framework. To the right, there's a section for RSS feeds for various categories like Everything: The Firehose, Blogs RSS, Buzz RSS, In Action RSS, News RSS, and Videos RSS. Below the main content, there's a section for Latest ASP.NET MVC News with a list of recent news items. The bottom right corner features a section for Bloglines with a list of recent blog posts.

DotNetKicks.com: Articles tagged with ASP.NET MVC

The DotNetKicks.com web site offers a collection of various links to interesting and actual ASP.NET MVC blog articles, tutorials, and so on.

URL: <http://www.dotnetkicks.com/tags/ASPNETMVC>

The screenshot shows the DotNetKicks.com website interface. At the top, there's a navigation bar with links like 'home', 'tags', 'ASPNETMVC', 'login', 'register', 'submit a story', 'upcoming stories', 'about', and 'blog'. Below the navigation bar, there's a banner for 'SAP Deployments' and a 'Stories recently tagged with 'ASPNETMVC'' section. This section lists three articles:

- ASP.NET MVC options for consolidating HTML** by James 5 days, 10 hours ago. The article discusses duplicating HTML bits and pieces of final rendered pages in ASP.NET MVC applications and offers solutions like Master Pages, custom controls, and composite controls. It also mentions the potential for duplicate HTML in WebForms land.
- Use MetadataType Attribute with ASP.NET MVC xVal Validation Framework** by gonale.wordpress.com 5 days, 22 hours ago. The article discusses incorporating the MetadataType attribute to allow usage with LINQ2SQL classes and mentions the xVal validation framework.
- Learn ASP.NET MVC (Getting Started & Digging Deeper)** by gonale.wordpress.com 5 days, 23 hours ago.

On the right side of the page, there are several advertisements and a search bar. The ads include '3607 hacked!', 'Looking for a .NET or Ruby on Rails? I'm available for hire - gailjoyce@gmail.com', 'Geminl. Project Control for Developers!', 'CounterSoft', 'Simply brilliant issue management for Software Development Teams FREE 5 User License', and 'SubSonic'.

Blogs

The following list includes URLs of blogs that contain a lot of interesting articles on the ASP.NET MVC framework:

- Scott Guthrie: <http://weblogs.asp.net/Scottgu/>
- Phil Haack: <http://www.haacked.com/>
- Scott Hanselman: <http://www.hanselman.com/blog/>
- Stephen Walther: <http://weblogs.asp.net/StephenWalther/>
- Brad Wilson: <http://bradwilson.typepad.com/blog/>
- Maarten Balliauw: <http://blog.maartenballiauw.be>
- Chris van de Steeg: <http://www.chrisvandesteeg.nl/>
- Troy Goode: <http://www.squaredroot.com>

Open source projects providing additional features for the ASP.NET MVC framework

This topic lists some open source projects that provide additional features for the ASP.NET MVC framework. These projects can be used when developing your own ASP.NET MVC web application. They can also facilitate development as well as provide additional features.

ASP.NET MVC Design Gallery

URL: <http://www.asp.net/mvc/gallery>

The ASP.NET MVC Design Gallery hosts free HTML design templates that you can download and easily use with your ASP.NET MVC applications. Each design template includes a `Site.master` file, a CSS stylesheet, and optionally, a set of images, partial views, and helper methods that support them.

MVC Contrib

URL: <http://www.codeplex.com/MVCContrib>

This project provides a series of assemblies that add functionality to the ASP.NET MVC framework. A number of useful base classes and extensions are provided.

Some of the included items are:

- Extra view helpers
- Extra ActionResult implementations
- IoC container controller factories for the popular containers: StructureMap, Windsor, Spring.Net, Unity, and so on
- Code snippets

xVal validation framework

URL: <http://xval.codeplex.com/>

xVal is a validation framework for ASP.NET MVC applications. xVal makes it easy to link up your choice of server-side validation mechanism with your choice of client-side validation library, neatly fitting both into the ASP.NET MVC architecture and conventions.

ASP.NET MVC Membership Starter Kit

URL: <http://www.codeplex.com/MvcMembership>

This project provides a base web site and class library that extends the default ASP.NET MVC framework project template with user and role administration, OpenID authentication, Windows Live authentication, and so on.

XForms

URL: <http://www.codeplex.com/mvcxforms>

ASP.NET MVC XForms is a simple, strongly-typed, extensible UI framework based on the W3C XForms spec. It provides a base set of form controls that allow the update of any complex model object, including complex nested lists. It uses clean, semantic HTML, and a fluent, lambda-based API.

jQuery for ASP.NET MVC

URL: <http://www.codeplex.com/jquery4mvc>

This project offers a set of helper methods that allow easier development of web applications combining the ASP.NET MVC framework and jQuery.

Simple ASP.NET MVC controls

URL: <http://www.codeplex.com/simplemvccontrols>

The simple ASP.NET MVC controls library helps to create an MVC web page more easily and without changing the context of the view. It currently features a code-driven fluent interface for creating an HTML form containing different sorts of controls and validation.

Alternative view engines

This topic lists some alternatives to the default view engine that is a part of the ASP.NET MVC framework.

Spark view engine

URL: <http://dev.dejardin.org/documentation>

Spark is a view engine for the ASP.NET MVC framework. It allows the use of pure HTML markup to create a view, rather than a combination of HTML and code.

NHaml view engine

URL: <http://weblogs.asp.net/stephenwalther/archive/2008/08/20/asp-net-mvc-tip-35-use-the-nhaml-view-engine.aspx>

The NHaml view engine is an alternative view engine for the ASP.NET MVC framework, providing a verbose language to render a view. It allows you to create a view without having to write a lot of code.

NVelocity view engine

URL: <http://weblogs.asp.net/stephenwalther/archive/2008/07/14/asp-net-mvc-tip-19-use-the-nvelocity-view-engine.aspx>

NVelocity is an alternative view engine to the ASP.NET MVC framework that provides an easy-to-use template language focused on building HTML pages, using a clean templating syntax.

Example ASP.NET MVC applications

This topic lists some example applications that have been built using the ASP.NET MVC framework. These applications can be used as a reference for creating your own applications.

MVC storefront

URL: <http://blog.wekeroad.com/mvc-storefront/> and <http://www.codeplex.com/mvcsamples>

This includes Rob Conery's set of screencasts and an example application featuring an online store written using ASP.NET MVC, Windsor, Linq to SQL, and so on.

FlickrExplorer

URL: <http://www.codeplex.com/FlickrXplorer>

flickrExplorer is an open source initiative to present users with a fast photo explorer and search tool that they can use to browse millions of photos in flickr. The application also demonstrates the use of jQuery in an ASP.NET MVC application.

Yonkly

URL: <http://www.codeplex.com/yonkly>

Yonkly is an open source Twitter clone, written in the ASP.NET MVC framework.

Kigg

URL: <http://www.codeplex.com/Kigg>

Kigg is an open source Digg clone developed using the ASP.NET MVC framework, LINQ to SQL, and ASP.NET AJAX.

CarTrackr

URL: <http://www.codeplex.com/CarTrackr>

CarTrackr is a sample application for the ASP.NET MVC framework that uses the repository pattern, and dependency injection using the Unity application block. It is an online software application that is designed to help you understand and track your fuel use and kilometers driven.

Index

Symbols

\$ 153
\$(function() { // ... } 153
[Bind] attribute 42
[OutputCache] attribute 129
[TestMethod] attribute 167

A

AcceptVerbsAttribute 68
AccountController 168, 169
AccountController constructor 170
action filters, ASP.NET MVC framework
 about 75
 cross-cutting concern 75
 IActionFilter interface 76
 IAuthorizationFilter interface 75
 IExceptionFilter interface 76
 IResultFilter interface 76
ActionLink method 35
action method 150
action method attributes, controller
 AcceptVerbsAttribute 68
 ActionNameAttribute 68
 AuthorizeAttribute 67
 HandleErrorAttribute 67
 NonActionAttribute 67
 OutputCacheAttribute 67
 ValidateAntiForgeryToken 68
ActionNameAttribute 68
ActionResult 22
ActionResult, creating
 ExecuteResult method 101, 102
 HtmlHelper extension method 104
 ImageResult method 102, 103
 ShowTitle action method 104

ActionResult, types

 Content 61
 ContentResult 61
 EmptyResult 61
 File 61
 FilePathResult 61
 FileStreamResult 61
 JavaScript 61
 JavaScriptResult 61
 Json 61
 JsonResult 61
 PartialView 61
 PartialViewResult 61
 Redirect 61
 RedirectResult 61
 RedirectToAction 61
 RedirectToRoute 61
 RedirectToRouteResult 61
 View 61
 ViewResult 61
ActionSelectionAttribute 63
Active Server Pages. *See* ASP
AddMultiple method 173
Ajax.BeginForm method 149
AJAX frameworks
 about 145, 146
 ASP.NET AJAX framework 147
 JavaScript Object Notation (JSON) 146
 XMLHttpRequest 146
AjaxHelper object, methods
 ActionLink 147
 BeginForm 147
 RouteLink 147
AjaxOptions, property
 confirm 148
 HttpMethod 148

- InsertionMode 148
- LoadingElementId 148
- OnBegin 148
- OnComplete 148
- OnFailure 148
- OnSuccess 148
- UpdateTargetId 148
- Url 148
- AntiForgeryToken method 35**
- Application_Start() event handler 19**
- ASP**
 - disadvantages 10
- ASP.NET AJAX framework**
 - about 147
 - ASP.NET MVC AJAX helper 147
- ASP.NET framework**
 - ASP.NET provider model 210-212
- ASP.NET framework, portals**
 - ASP.NET MVC 225
 - Aspdotnetmvc.com 227
 - Dotnetkicks.com 228
- ASP.NET MVC**
 - and ASP.NET Webforms, combining 135
 - and ASP.NET Webforms, differences 165
- ASP.NET MVC AJAX helper**
 - ActionLink method 147
 - AjaxHelper object, methods 147
 - AjaxOptions, property 148
 - BeginForm method 147
 - JsonResult, working with 150
 - methods 147
 - RouteLink method 147
- ASP.NET MVC application**
 - views, building at compile time 142
- ASP.NET MVC applications, example**
 - CarTrackr 232
 - FlickrExplorer 232
 - Kigg 232
 - MVC storefront 231
- ASP.NET MVC Design Gallery, open source project 229**
- ASP.NET MVC framework**
 - advantages 12
 - and ASP.NET Webforms 12, 13
 - and ASP.NET Webforms, selecting criteria 13, 14
 - and WCSF differences, URL 11
 - blog, URLs 228, 229
 - control, creating 92-95
 - culture preferences, setting 132-134
 - custom ActionResult, creating 101
 - filter attribute, creating 96-100
 - goals 10, 11
 - language preferences, setting 132-134
 - open source projects 229
 - output caching 127
 - SimpleViewEngine 112
 - TempData 119
 - ViewEngine, creating 105
 - view engines 231
- ASP.NET MVC framework, extending**
 - control, creating 92-95
 - custom ActionResult, creating 101
 - filter attribute, creating 96-100
- ASP.NET MVC Membership Starter Kit 207, 208**
- ASP.NET MVC Membership Starter Kit, open source project 230**
- ASP.NET MVC mock helpers 217**
- ASP.NET MVC request flow. *See* ASP.NET MVC request life cycle**
- ASP.NET MVC request life cycle**
 - controller execution 54
 - diagram 52
 - extensibility 54
 - IHandler, created by route handler 53
 - IHandler, determined by controller 53
 - RouteTable, creating 52
 - routing engine, route determined by 53
 - steps 52
 - UrlRoutingModule, request intercepted by 53
 - view, rendering 54
 - ViewEngine, creating 54
- ASP.NET MVC request life cycle, extensibility**
 - controller 55
 - ControllerFactory 55
 - IView 56
 - MvcRouteHandler 55
 - route objects 54
 - ViewEngine 55, 56
- ASP.NET MVC web application**
 - controller 60

- data, validating 45
- file uploads, handling 39
- form, creating 33
- hosting 186
- IIS versions 184
- model 56
- ModelBinder attribute, using 40
- platforms, for running 184
- posts, handling 37
- running, IIS versions 184
- running, platforms 183
- view 68
- ASP.NET MVC web application, hosting**
 - route table, modifying 189-191
 - wildcard script map, creating in Internet Information Services (IIS) 6.0 188, 189
 - wildcard script map, creating in Internet Information Services (IIS) 7.0 187
- ASP.NET MVC web application project**
 - ActionResult 22
 - content folder 18
 - controllers folder 18
 - creating 16, 18
 - Employee controller 20
 - EmployeeController class 20
 - EmployeeController class, coding 21, 22
 - firstname parameter 20
 - folders 18
 - models folder 18
 - MvcApplication1 project, code 19
 - routing 19
 - scripts folder 18
 - Show.aspx page, creating 22, 23
 - Show action 20
 - ViewData used 23
 - views folder 18
- ASP.NET provider model 210-212**
- ASP.NET routing**
 - about 79
 - versus URL rewriting 80
- ASP.NET web application**
 - <authentication> element 125
 - authentication 120
 - authentication options 125
 - authorization 120
 - caching 127
 - globalization 129
 - membership 120
 - role based security, implementing in controller 122
 - routes, testing 178, 179
 - user based security, implementing in controller 122
 - web site administration tool 121
 - web site security, configuring 122
 - web site security configuring, web site administration tool used 121
- ASP.NET Webforms**
 - advantages 12
 - and ASP.NET MVC, combining 135
 - and ASP.NET MVC framework 12, 13
 - and ASP.NET MVC framework, selecting criteria 13, 14
- ASP.NET Webforms and ASP.NET MVC, combining**
 - about 135
 - ASP.NET, plugging in existing ASP.NET MVC application 139, 140
 - ASP.NET MVC, plugging in existing ASP.NET application 135-138
 - data, sharing 140, 141
 - web.config file, enabling 136
- ASP.NET web site administration tool**
 - web site security, configuring 121, 122
- ASP.NET MVC web application, hosting**
 - wildcard script map, creating in Internet Information Services (IIS) 6.0 188, 189
- AttributeEncode method 35**
- AuthenticatedRouteConstraint class 85**
- authentication 120**
- authentication options**
 - cookieless 125
 - defaultUrl 125
 - domain 126
 - loginUrl 126
 - mode 125
 - name 126
 - path 126
 - protection 126
 - requireSSL 126
 - slidingExpiration 126
 - timeout 126
- authorization 120**

AuthorizeAttribute 67
AuthorizeAttribute parameters 122

B

BeginForm method 35
blog URLs, ASP.NET MVC framework 228, 229
Button method 35

C

caching
 [OutputCache] attribute 129
 about 127
 client-side caching 127
 OutputCacheLocation.None 129
 output caching 127
 server-side caching 127
CalculationEngine constructor 173
CarController class 208, 209
CarRepository implementation 201
CarRepository instance 205
CarTrackr
 car class 202
 car details 197, 198
 controllers, building 203
 data layer 199
 form validation 208-210
 home page 194
 ICarRepository interface 201
 IUserRepository 203
 list of cars 196, 197
 login screen 195
 mock repository 214, 215
 RefuellingController 203
 refuellings list 198
 unit testing 213, 214
CGI 9
CheckBox method 35
click() event handler 160
Common Gateway Interface. *See* CGI
Content 61
ContentResult 61
control creating, ASP.NET MVC framework
 about 92
 DisplayEmployee.ascx.cs 94
 Employee class, creating 92

 ViewData type 95
 ViewUserController class 94
controller
 about 8, 9, 60
 action method attributes 67
 action method selection 63
 creating 60, 61
 data, reading from request 62
 data, rendering 61
 HandleUnknownAction method 64-66
 unknown controller actions, handling 64
controller, unit testing 27-29
ControllerActionInvoker 63
ControllerBuilder 86
ControllerContext 63
CreatePartialView() method, overriding 106
CreateView() method, overriding 106
cross-cutting concerns
 about 96
 example 96
CultureInfo instance 134
culture preferences, setting 132-134
CurrentCulture property 134
custom ActionResult, creating 101

D

data, validating 45, 46
data layer, CarTrackr
 LINQ to SQL model 200, 201
 repository pattern 201, 202
DefaultNamespaces property 86
dependency injection
 about 202
 car class 202
 CarTrackr controllers, building 203, 204
 unity, using 204-206
div element 160
DropDownList method 35

E

e.preventDefault() 156
Echo action method 149
EchoTarget div element 149
EmptyResult 61
Encode method 35
EnsureValid method 209

EvalBoolean method 35
EvalString method 35
ExecuteResult() method 101

F

File 61
FileContentResult 61
FilePathResult 61
FileStreamResult 61
file uploads, handling
 upload controller action, creating 39
FilterAttribute class 96
filter attribute creating, ASP.NET MVC
 framework 96, 97
 AttributeUsage attribute 97
 LoggingAttribute class 97
 LogName property 97
find() 163
foreach loop 73
form, creating
 approaches 33
 HtmlHelper used 34
 HTML used 33, 34
FormCollection 180, 181
formsAuthenticationMock 175

G

get_object() method 151
GetRouteData method 81, 179
GetRuleViolations method 209
globalization
 global resources, using 132
 local resources, using 130-132
 resources 129
global resources 132

H

HandleErrorAttribute 67
HandleUnknownAction method, controller
 64, 66
Hidden method 35
Home controller 19
HomeController class 112
HomeControllerTest class 167, 168
HtmlHelper.ActionLink method 73

HtmlHelper.ValidationMessage() 59
HtmlHelper.ValidationSummary method
 48

 about 34, 35
 ActionLink method 35
 AntiForgeryToken method 35
 AttributeEncode method 35
 BeginForm method 35
 Button method 35
 CheckBox method 35
 DropDownList method 35
 Encode method 35
 EvalBoolean method 35
 EvalString method 35
 Hidden method 35
 Image method 35
 ListBox method 35
 Mailto method 35
 NavigateButton method 35
 Password method 35
 RenderAction method 35, 36
 RenderPartial method 36
 RouteLink method 35, 36
 SubmitButton method 36
 SubmitImage method 36
 TextArea method 36
 TextBox method 35
 ValidationMessage method 36
 ValidationSummary method 36

HtmlHelper extension method 104

HttpRequest object 39, 40

I

IActionFilter interface, action filters
 OnActionExecuted method 76
 OnActionExecuting method 76
IAuthorizationFilter interface, action filters
 75
ICalculator interface 173
ICarRepository interface 201
IControllerFactory interface 55
IController interface 55
IDataErrorInfo 59
IExceptionHandler interface, action filters 76
IFormsAuthentication implementation
 170, 172

- IHandler 53**
- IIS 6.0 188**
- IIS 7.0 187**
- IIS versions 184**
- Image method 35**
- ImageResult class**
 - ImageFormat property 103
 - Image property 103
 - ImageResult constructor 103
 - properties 103
- ImageResult constructor 103**
- IModelBinder implementations**
 - about 45
 - DefaultModelBinder class 45
 - FormCollectionModelBinder class 45
- Index action 62**
- internationalization 129**
- InternationalizationAttribute 133, 134**
- Internet Information Services (IIS) 7.0**
 - wildcard script map, creating 187
- Internet Information Services (IIS) 7.0**
 - integrated mode and classic mode, differences 184-186
- IRefuellingRepository 204**
- IResultFilter interface, action filters**
 - methods 76
 - OnResultExecuted method 76
 - OnResultExecuting method 76
- IRouteConstraint interface 84**
- IRouteHandler**
 - GetHandler method 81
- IUnityContainer instance 205**
- IUserRepository 203**
- IValueProvider 180**
- IValueProvider instance 42**
- IValueProvider interface 42**
- IView 56**
- IViewEngine interface, implementing 105**
- IView interface 107**

J

- JavaScript 61**
- JavaScript Object Notation. *See* JSON**
- JavaScriptResult 61**
- jQuery, AJAX framework**
 - about 152

- jQuery UI, using 159-163
- JsonResult, working with 157
- syntax 153, 154
- using, with ASP.NET MVC 154, 155
- jQuery for ASP.NET MVC, open source project 230**
- JSON 146**
- Json 61**
- JsonResult 61**

L

- language preferences, setting 132-134
- LINQ to SQL model 200**
- ListBox method 35**
- localization 129**
- local resources 130, 131**
- LoggingAttribute class 97, 99**
- login action method, testing 174-176**
- LogName property 97, 98**
- Logout action method 213**

M

- Mailto method 35**
- MapRoute method**
 - arguments 83
- membership 120**
- MembershipProvider instance 171, 172**
- Microsoft.Web.Mvc 104**
- mocking frameworks**
 - EasyMock 172
 - Moq 172
 - Rhino Mocks 172
 - TypeMock 172
- model**
 - about 8, 56
 - creating 56
 - Task class 57
 - validation, enabling 58, 60
 - ViewData class 58, 60
- model-view-controller. *See* MVC pattern**
- model-view-presenter. *See* MVP pattern**
- ModelBinder 180**
- ModelBinder attribute**
 - Contact class 43
 - custom ModelBinder, creating 43-45

- default ModelBinder, using 41-43
- DefaultModelBinder class 44
- IValueProvider interface 42
- using 40
- ModelBinders.Binders.Add()** method 43
- ModelBindingContext** parameter 45
- ModelState** dictionary 210
- Moq**, mocking framework 172, 220-222
- MVC Contrib**, open source project 229
- MvcMockHelpers** class, extension methods 177, 178
- MVC pattern**
 - about 7-9
 - vs MVP pattern 11
- MVP pattern**
 - vs MVC pattern 11

N

- NavigateButton** method 35
- NHaml** view engine 231
- NonActionAttribute** 67
- NVelocity** view engine 231

O

- OnActionExecuted** method, **IActionFilter** interface 98
- OnActionExecuting** method, **IActionFilter** interface 98
- OnJsonSampleColorsCompleted** JavaScript method 151
- open source projects, ASP.NET MVC framework**
 - ASP.NET MVC Design Gallery 229
 - ASP.NET MVC Membership Starter Kit 230
 - jQuery for 230
 - MVC Contrib 229
 - simple ASP.NET MVC controls 230
 - XForms 230
 - xVal validation framework 230
- OutputCacheAttribute** 67
- OutputCacheAttribute**, parameters
 - CacheProfile attribute 128
 - Duration attribute 128
 - Location attribute 128
 - NoStore attribute 128
 - SqlDependency attribute 128

- VaryByContentEncoding attribute 128
- VaryByCustom attribute 128
- VaryByHeader attribute 128
- VaryByParam attribute 128
- OutputCacheLocation.None** 129
- output caching** 127

P

- PartialView** 61
- PartialViewResult** 61
- Password** method 35
- posts, handling**
 - about 37
 - action method parameters 38
 - request variables 37
 - request variables, objects updating from 37, 38

R

- Redirect** 61
- RedirectResult** 61
- RedirectToAction** 61
- RedirectToRoute** 61
- RedirectToRouteResult** 61
- RefuellingController** 203
- RegisterRoutes** method 179
- Remove** action method 214, 215
- Render()** method 108
- RenderAction** method 35, 36
- RenderPartial** method 36, 74
- Resolve()** method 109
- resources, globalization**
 - about 129
 - global resources 132
 - local resources 130, 131
- RetrieveJsonSampleColors** JavaScript method 151
- RhinoMocks**, mocking framework 217-219
- RouteCollection** 179
- RouteCollection** class, **UrlRoutingModule** 81
- RouteLink** method 35, 36
- routes**
 - ASP.NET MVC and ASP.NET, combining 88
 - catch-all routes 85

- ControllerBuilder 86
- DefaultNamespaces property 86
- defining 82
- example, Global.asax file 82
- MapRoute method 83
- parameter constraints 84
- routing namespaces 86, 87
- StopRouteHandler() 88
- URLs, creating 89
- routes, testing**
 - about 178
 - ArchiveRoute testing, MvcMockHelpers
 - extension methods used 179
 - GetRouteData method 179
 - RouteCollection 179
- routing 19**
- routing patterns**
 - {controller}/{action}/{id} 82
 - {controller}/{action}/{id}.aspx 82
 - {department}/{title}.aspx 82
 - {language}-{country}/{controller}/{action}/{id} 82
 - archive/{year}-{month}/{title}.aspx 82
- RuleViolation 209**
- S**
- SaveAs() method 39**
- session state, ASP.NET MVC framework**
 - about 116
 - configuring 117
 - Message key 117
 - session data, reading 117
 - session data, writing 117
 - sessionstate element, attributes 118
- sessionstate element, attributes**
 - cookieless attribute 119
 - cookieName attribute 118
 - regenerateExpiredSessionId attribute 119
 - timeout attribute 118
- SetFakeControllerContext extension method 213**
- show() method 161**
- ShowTitle action method 104**
- simple ASP.NET MVC controls, open**
 - source project 230

- SimpleView**
 - code 109-112
- SimpleViewEngine 112**
- SiteMapPath control 212**
- spark view engine 231**
- StopRouteHandler() 88**
- strong-typed view data**
 - about 24
 - ViewData.Model property 24, 25
 - ViewData object 24
- SubmitButton method 36**
- SubmitImage method 36**
- Sys.Net.WebRequest class 151**

T

- T4 71**
- Task class 57**
- TDD 13, 166**
- TempData, ASP.NET MVC framework 119**
- Test-Driven Development. *See* TDD**
- TextArea method 36**
- TextBox method 35**
- Text Template Transformation Toolkit.**
 - See* T4
- TryUpdateModel method 179, 180**
- TypeMock, mocking framework 222-224**

U

- units 27**
- unit testing**
 - about 27, 166
 - AccountController constructor 170
 - action method, testing 170
 - AddMultiple method 173
 - benefits 166
 - example 166-168
 - for controller 27-29
 - formsAuthenticationMock 175
 - frameworks 166
 - generating 168-170
 - login action method, testing 174-176
 - mocking frameworks 172
- unit testing, CarTrackr 213, 214**
 - about 213, 214
 - mock repository 214

- UpdateModel** 38
- UpdateModel method** 179, 180
- UpdateModel scenarios, testing**
 - UpdateModel (or TryUpdateModel) 179
 - UpdateModel method (or TryUpdateModel method) 180, 181
- UpdateModelStateWithViolations method** 210
- URL rewriting**
 - versus ASP.NET routing 80
- UrlRoutingModule, ASP.NET routing**
 - about 80, 81
 - IHttpHandler 81
 - IRouteHandler 81
 - RouteCollection class 81
- URLs, creating from routes** 89

V

- ValidateAntiForgeryToken** 68
- ValidationMessage method** 36
- ValidationSummary method** 36
- VaryByContentEncoding** 127
- VaryByCustom** 127
- VaryByHeader** 127
- VaryByParam** 127
- View** 61
- view**
 - about 8, 68
 - creating 70, 71
 - location 69, 70
 - master pages 71
 - partial views 74
 - RenderPartial method 74
 - view markup 72
- view, adding** 26
- ViewData.Model property** 24, 25
- ViewData.ModelState.AddModelError method** 47
- ViewData.ModelState collection** 49
- ViewData class** 58

- ViewData dictionary** 138
- ViewData object** 24
- ViewData property** 94
- ViewData type** 95
- ViewEngine** 56
 - CreatePartialView() method, overriding 106
 - CreateView() method, overriding 106
 - IViewEngine interface, implementing 105
 - IView interface 107, 108
 - Render() method, implementing 108
 - Resolve() method 109
 - SimpleViewEngine, creating 106, 107
- view engines, ASP.NET MVC framework**
 - NHaml view engine 231
 - NVelocity view engine 231
 - spark view engine 231
- ViewResult** 61
- views, building at compile time** 142
- ViewUserControl class** 94

W

- WCSF**
 - and ASP.NET MVC differences, URL 11
- Web Client Software Factory.** *See* WCSF
- web security, configuring** 121, 122
- wildcard script map**
 - creating in Internet Information Services (IIS) 6.0 188, 189
 - creating in Internet Information Services (IIS) 7.0 187

X

- XForms, open source project** 230
- XMLHttpRequest** 146
- XRDS action method** 207
- xVal validation framework, open source project** 230



Thank you for buying
ASP.NET MVC 1.0 Quickly

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

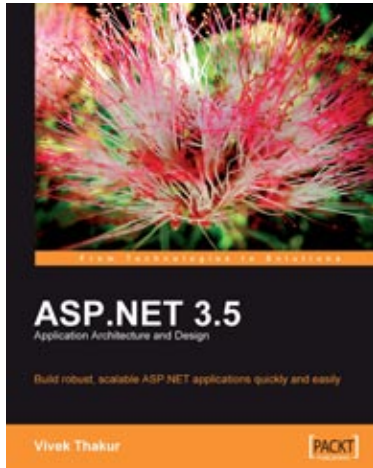
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



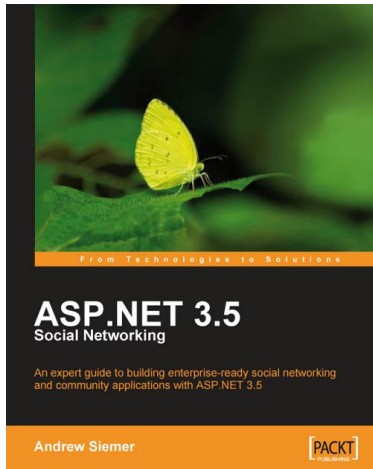
ASP.NET 3.5 Application Architecture and Design

ISBN: 978-1-847195-50-0

Paperback: 239 pages

Build robust, scalable ASP.NET applications quickly and easily

1. Master the architectural options in ASP.NET to enhance your applications
2. Develop and implement n-tier architecture to allow you to modify a component without disturbing the next one
3. Design scalable and maintainable web applications rapidly
4. Implement ASP.NET MVC framework to manage various components independently



ASP.NET 3.5 Social Networking

ISBN: 978-1-847194-78-7

Paperback: 556 pages

An expert guide to building enterprise-ready social networking and community applications with ASP.NET 3.5

1. Create a full-featured, enterprise-grade social network using ASP.NET 3.5
2. Learn key new ASP.NET topics in a practical, hands-on way: LINQ, AJAX, C# 3.0, n-tier architectures, and MVC
3. Build friends lists, messaging systems, user profiles, blogs, message boards, groups, and more
4. Rich with example code, clear explanations, interesting examples, and practical advice — a truly hands-on book for ASP.NET developers

Please check www.PacktPub.com for information on our titles