Microsoft

OFFICIAL MICROSOFT LEARNING PRODUCT

20486B

Developing ASP.NET MVC 4 Web Applications

Product Number: 20486B

Part Number : X18-52163

Released: 05/2013

MICROSOFT LICENSE TERMS
OFFICIAL MICROSOFT LEARNING PRODUCTS
MICROSOFT OFFICIAL COURSE Pre-Release and Final Release Versions

---

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the Licensed Content named above, which includes the media on which you received it, if any. These license terms also apply to any updates, supplements, internet based services and support services for the Licensed Content, unless other terms accompany those items. If so, those terms apply.

BY DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT DOWNLOAD OR USE THE LICENSED CONTENT.

If you comply with these license terms, you have the rights below.

---

1. **DEFINITIONS.**

   a. "Authorized Learning Center" means a Microsoft Learning Competency Member, Microsoft IT Academy Program Member, or such other entity as Microsoft may designate from time to time.

   b. "Authorized Training Session" means the Microsoft-authorized instructor-led training class using only MOC Courses that are conducted by a MCT at or through an Authorized Learning Center.

   c. "Classroom Device" means one (1) dedicated, secure computer that you own or control that meets or exceeds the hardware level specified for the particular MOC Course located at your training facilities or primary business location.

   d. "End User" means an individual who is (i) duly enrolled for an Authorized Training Session or Private Training Session, (ii) an employee of a MPN Member, or (iii) a Microsoft full-time employee.

   e. "Licensed Content" means the MOC Course and any other content accompanying this agreement. Licensed Content may include (i) Trainer Content, (ii) software, and (iii) associated media.

   f. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program, and (iii) holds a Microsoft Certification in the technology that is the subject of the training session.

   g. "Microsoft IT Academy Member" means a current, active member of the Microsoft IT Academy Program.

   h. "Microsoft Learning Competency Member" means a Microsoft Partner Network Program Member in good standing that currently holds the Learning Competency status.

   i. "Microsoft Official Course" or "MOC Course" means the Official Microsoft Learning Product instructor-led courseware that educates IT professionals or developers on Microsoft technologies.

j. "Microsoft Partner Network Member" or "MPN Member" means a silver or gold-level Microsoft Partner Network program member in good standing.

k. "Personal Device" means one (1) device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular MOC Course.

l. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.

m. "Trainer Content" means the trainer version of the MOC Course and additional content designated solely for trainers to use to teach a training session using a MOC Course. Trainer Content may include Microsoft PowerPoint presentations, instructor notes, lab setup guide, demonstration guides, beta feedback form and trainer preparation guide for the MOC Course. To clarify, Trainer Content does not include virtual hard disks or virtual machines.

2. **INSTALLATION AND USE RIGHTS**. The Licensed Content is licensed not sold. The Licensed Content is licensed on a one copy per user basis, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

2.1 Below are four separate sets of installation and use rights. Only one set of rights apply to you.

a. **If you are a Authorized Learning Center:**

i. If the Licensed Content is in digital format for each license you acquire you may either:

1. install one (1) copy of the Licensed Content in the form provided to you on a dedicated, secure server located on your premises where the Authorized Training Session is held for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session, **or**
2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session.

ii. You agree that:

1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
3. for all Authorized Training Sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,

5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of the Authorized Training Session,
6. you will only provide access to the Licensed Content to End Users and MCTs,
7. you will only provide access to the Trainer Content to MCTs, and
8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.

b. **If you are a MPN Member.**

i. If the Licensed Content is in digital format for each license you acquire you may either:

1. install one (1) copy of the Licensed Content in the form provided to you on (A) one (1) Classroom Device, or (B) one (1) dedicated, secure server located at your premises where the training session is held for use by one (1) of your employees attending a training session provided by you, or by one (1) MCT that is teaching the training session, **or**
2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for use by one (1) End User attending a Private Training Session, or one (1) MCT that is teaching the Private Training Session.

ii. You agree that:

1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
3. for all training sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,
5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of each training session,
6. you will only provide access to the Licensed Content to End Users and MCTs,
7. you will only provide access to the Trainer Content to MCTs, and
8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.

c. **If you are an End User:**

You may use the Licensed Content solely for your personal training use. If the Licensed Content is in digital format, for each license you acquire you may (i) install one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install another copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content; or (ii) print one (1) copy of the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.

d. **If you are a MCT.**

i. For each license you acquire, you may use the Licensed Content solely to prepare and deliver an Authorized Training Session or Private Training Session. For each license you acquire, you may install and use one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.

ii. **Use of Instructional Components in Trainer Content.** You may customize, in accordance with the most recent version of the MCT Agreement, those portions of the Trainer Content that are logically associated with instruction of a training session. If you elect to exercise the foregoing rights, you agree: (a) that any of these customizations will only be used for providing a training session, (b) any customizations will comply with the terms and conditions for Modified Training Sessions and Supplemental Materials in the most recent version of the MCT agreement and with this agreement. For clarity, any use of "*customize"* refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

2.2 **Separation of Components.** The Licensed Content components are licensed as a single unit and you may not separate the components and install them on different devices.

2.3 **Reproduction/Redistribution Licensed Content**. Except as expressly provided in the applicable installation and use rights above, you may not reproduce or distribute the Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.

2.4 **Third Party Programs**. The Licensed Content may contain third party programs or services. These license terms will apply to your use of those third party programs or services, unless other terms accompany those programs and services.

2.5 **Additional Terms**. Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to that respective component and supplements the terms described in this Agreement.

3. **PRE-RELEASE VERSIONS.** If the Licensed Content is a pre-release ("**beta**") version, in addition to the other provisions in this agreement, then these terms also apply:

a. **Pre-Release Licensed Content.** This Licensed Content is a pre-release version. It may not contain the same information and/or work the way a final version of the Licensed Content will. We may change it for the final version. We also may not release a final version. Microsoft is under no obligation to provide you with any further content, including the final release version of the Licensed Content.

b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software, technologies, or products to third parties because we include your feedback in them. These rights

survive this agreement.

c. **Term**. If you are an Authorized Training Center, MCT or MPN, you agree to cease using all copies of the beta version of the Licensed Content upon (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) sixty (60) days after the commercial release of the Licensed Content, whichever is earliest ("**beta term**"). Upon expiration or termination of the beta term, you will irretrievably delete and destroy all copies of same in the possession or under your control.

4. **INTERNET-BASED SERVICES**. Microsoft may provide Internet-based services with the Licensed Content, which may change or be canceled at any time.

a. **Consent for Internet-Based Services.** The Licensed Content may connect to computer systems over an Internet-based wireless network. In some cases, you will not receive a separate notice when they connect. Using the Licensed Content operates as your consent to the transmission of standard device information (including but not limited to technical information about your device, system and application software, and peripherals) for internet-based services.

b. **Misuse of Internet-based Services.** You may not use any Internet-based service in any way that could harm it or impair anyone else's use of it. You may not use the service to try to gain unauthorized access to any service, data, account or network by any means.

5. **SCOPE OF LICENSE**. The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
   - install more copies of the Licensed Content on devices than the number of licenses you acquired;
   - allow more individuals to access the Licensed Content than the number of licenses you acquired;
   - publicly display, or make the Licensed Content available for others to access or use;
   - install, sell, publish, transmit, encumber, pledge, lend, copy, adapt, link to, post, rent, lease or lend, make available or distribute the Licensed Content to any third party, except as expressly permitted by this Agreement.
   - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation;
   - access or use any Licensed Content for which you are not providing a training session to End Users using the Licensed Content;
   - access or use any Licensed Content that you have not been authorized by Microsoft to access and use; or
   - transfer the Licensed Content, in whole or in part, or assign this agreement to any third party.

6. **RESERVATION OF RIGHTS AND OWNERSHIP**. Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content. You may not remove or obscure any copyright, trademark or patent notices that appear on the Licensed Content or any components thereof, as delivered to you.

7. **EXPORT RESTRICTIONS**. The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, End Users and end use. For additional information, see www.microsoft.com/exporting.

8. **LIMITATIONS ON SALE, RENTAL, ETC. AND CERTAIN ASSIGNMENTS**. You may not sell, rent, lease, lend or sublicense the Licensed Content or any portion thereof, or transfer or assign this agreement.

9. **SUPPORT SERVICES**. Because the Licensed Content is "as is", we may not provide support services for it.

10. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon any termination of this agreement, you agree to immediately stop all use of and to irretrievable delete and destroy all copies of the Licensed Content in your possession or under your control.

11. **LINKS TO THIRD PARTY SITES**. You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.

12. **ENTIRE AGREEMENT.** This agreement, and the terms for supplements, updates and support services are the entire agreement for the Licensed Content.

13. **APPLICABLE LAW.**
    a. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.

    b. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.

14. **LEGAL EFFECT**. This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

15. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS," "WITH ALL FAULTS," AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES GIVE NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS UNDER OR IN RELATION TO THE LICENSED CONTENT. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES EXCLUDE ANY IMPLIED WARRANTIES OR CONDITIONS, INCLUDING THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**

16. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. TO THE EXTENT NOT PROHIBITED BY LAW, YOU CAN RECOVER FROM MICROSOFT CORPORATION AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO USD$5.00. YOU AGREE NOT TO SEEK TO RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES FROM MICROSOFT CORPORATION AND ITS RESPECTIVE SUPPLIERS.**

This limitation applies to

- anything related to the Licensed Content, services made available through the Licensed Content, or content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 $ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.
Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence , aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Revised December 2011

Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.

- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance[1]. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.

- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning
www.microsoft.com/learning

Microsoft | Learning

[1] *IDC,* Value of Certification: Team Certification and Organizational Performance, *November 2006*

Acknowledgments

Microsoft Learning wants to acknowledge and thank the following for their contribution toward developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

Lakshmi Kamal—Lead Content Developer

Lakshmi is a practice lead for learning design at Sify and has close to seven years of work experience in instructional design and development for software-based and hardware–based training. Under Lakshmi's leadership from an instructional design perspective, the Sify learning design team has been able to execute over 400 hours of instructor-led training and e-learning for Microsoft Learning on their products and technologies such as Windows Server 2008, Microsoft Office SharePoint Server 2007, SQL Server 2008, Office Communications Server 2007, Windows Vista, Visual Studio 2008, and Office Server servers. Lakshmi has also been the Lead learning designer for projects for Dell, General Electric, Cisco, Oracle, Allianz, and Standard Charted Bank.

Lakshmi is conversant with authoring and development tools such as MS Office, Adobe Captivate, Snagit, and Articulate. She holds a Master of Philosophy degree and a Post Graduation degree in English.

Uma—Content Developer

Uma is a Senior Instructional Designer at Sify Technologies Ltd. She has over six years of work experience in instructional design and development. She has designed and developed several information technology–based courses for Microsoft, Oracle, Standard Charted Bank, Cisco, Dell, and Comcast.

Alistair Matthews—Subject Matter Expert

Alistair Mathews is a consultant with extensive and cutting-edge experience in Microsoft technologies, Alistair has spent the last ten years developing with, consulting on, and communicating about both Developer and IT Professional sides of SharePoint, Visual Studio, Active Directory, Exchange and Windows. He architects and delivers custom SharePoint solutions in knowledge management, enterprise and web content management. He now works with customers throughout Microsoft to document products and develop learning materials of all types.

Paul Litwin—Technical Reviewer

Paul Litwin is a developer who specializes in ASP.NET, C#, SQL Server, SQL Server Reporting Services, and related technologies. Paul is a Programming Manager with Fred Hutchinson Cancer Research Center in Seattle. He is also the owner of Deep Training, a developer-owned training company that specializes in .NET developer & SQL Server Reporting Services training.

Paul is also an author of a number of books and training material. These training contents are based on ASP.NET, SQL Server Reporting Services, and Micrsoft Access. Currently, Paul is co-writing Agile ASP.NET Unleashed along with Stephen Walther and Ruth Wather. Paul is the conference chair of Microsoft ASP.NET Connections, a Microsoft MVP, and a member of the INETA Speakers Bureau. He is also a regular presenter at a number of conferences, user groups, and code camps.

Contents

Lab Answer Keys

About This Course

This section provides a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This second release ('B') MOC version of course 20486B has been developed on RTM software. Microsoft Learning has released the 'B' version with enhanced PowerPoint slides, copy-edited content, and Course Companion content on Microsoft Learning site.

In this 5-day course, the professional web developers will learn to develop advanced ASP.NET MVC application using .NET Framework 4.5 tools and technologies. The focus will be on coding activities that enhance the performance and scalability of the Web site application. This course will also prepare the student for exam 70-486.

Audience

This course is intended for professional web developers who use Microsoft Visual Studio in an individual-based or team-based, small-sized to large development environment. Candidates for this course are interested in developing advanced web applications and want to manage the rendered HTML comprehensively. They want to create websites that separate the user interface, data access, and application logic.

Student Prerequisites

This course requires that you meet the following prerequisites:

- A minimum of two to three years of experience in developing web-based applications by using Microsoft Visual Studio and Microsoft ASP.NET.
- Proficiency in using the .NET Framework and some familiarity with the C# language.

Students who attend this training can meet the prerequisites by attending the following courses or by obtaining equivalent knowledge and skills:

- 20483A Programming in C#
- 10958A Programming Fundamentals of Web Applications

Course Objectives

After completing this course, students will be able to:

- Describe the Microsoft Web Technologies stack and select an appropriate technology to use to develop any given application.
- Design the architecture and implementation of a web application that will meet a set of functional requirements, user interface requirements, and address business models.

- Create MVC Models and write code that implements business logic within Model methods, properties, and events.
- Add Controllers to an MVC Application to manage user interaction, update models, and select and return Views.
- Create Views in an MVC application that display and edit data and interact with Models and Controllers.
- Run unit tests and debugging tools against a web application in Visual Studio 2012.
- Develop a web application that uses the ASP.NET routing engine to present friendly URLs and a logical navigation hierarchy to users.
- Implement a consistent look and feel, including corporate branding, across an entire MVC web application.
- Use partial page updates and caching to reduce the network bandwidth used by an application and accelerate responses to user requests.
- Write JavaScript code that runs on the client-side and utilizes the jQuery script library to optimize the responsiveness of an MVC web application.
- Implement a complete membership system in an MVC 4 web application.
- Build an MVC application that resists malicious attacks and persists information about users and preferences.
- Describe how to write a Windows Azure web service and call it from and MVC application.
- Describe what a Web API is and why developers might add a Web API to an application.
- Modify the way browser requests are handled by an MVC application.
- Describe how to package and deploy an ASP.NET MVC 4 web application from a development computer to a web server for staging or production.

Course Outline

The course outline is as follows:

Module 1, “Exploring ASP.NET MVC 4", provides an overview of the Microsoft web technologies stack. It also describes the three programming models available in ASP.NET 4.5 such as web pages, web forms, and MVC.

Module 2, “Designing ASP.NET MVC 4 Web Applications", explains the planning of the overall architecture, controllers, views, and models of the MVC 4 web application.

Module 3, “Developing ASP.NET MVC 4 Models", explains how to create Models in an MVC 4 web application. The module also describes how to implement a connection to a database using the Entity Framework.

Module 4, “Developing ASP.NET MVC 4 Controllers", explains how to create Controllers in an MVC 4 web application. The module also describes how to write code in action filters that executes before or after a controller action.

Module 5, “Developing ASP.NET MVC 4 Views", explains how to create Views in an MVC 4 application that display and edit data and interact with Models and Controllers.

Module 6, “Testing and Debugging ASP.NET MVC 4 Web Applications", explains how to unit test the components of an MVC 4 application and to implement exception handling strategy for the application.

Module 7, “Structuring ASP.NET MVC 4 Web Applications", explains how to structure an MVC 4 web application that uses routing engine and navigation controls.

Module 8, “Applying Styles to ASP.NET MVC 4 Web Applications", explains how to implement a consistent look and feel to an MVC application by using a template view.

Module 9, "Building Responsive Pages in ASP.NET MVC 4 Web Applications" explains how to use partial page updates and caching to build responsive pages in an MVC 4 web application.

Module 10, "Using JavaScript and jQuery for Responsive MVC 4 Web Applications", explains how to increase the responsiveness of the MVC 4 web application by using JavaScript and jQuery.

Module 11, "Controlling Access to ASP.NET MVC 4 Web Applications", explains how to implement authentication and authorization for accessing the MVC 4 web application.

Module 12, "Building a Resilient ASP.NET MVC 4 Web Application", explains how to build an MVC 4 web application that is stable, reliable, resists malicious attacks, and persists information about users and preferences.

Module 13, "Using Windows Azure Web Services in ASP.NET MVC 4 Web Applications", provides an overview on Windows Azure and also explains how to design and write a Windows Azure service.

Module 14, "Implementing Web APIs in ASP.NET MVC 4 Web Applications", provides an overview of Web API and explains how to develop a Web API and call it from other applications.

Module 15, "Handling Requests in ASP.NET MVC 4 Web Applications", describes how to create HTTP Modules and HTTP Handlers to handle requests in an MVC 4 web application. The module also describes the use of Web Sockets in creating a live data feed in the MVC 4 web application.

Module 16, "Deploying ASP.NET MVC 4 Web Applications", explains how to deploy a completed MVC application to a web server or Windows Azure.

Course Materials

The following materials are included with your kit:

- ***Course Handbook:*** a succinct classroom learning guide that provides the critical technical information in a crisp, tightly-focused format, which is essential for an effective in-class learning experience.
  - **Lessons**: guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
  - **Labs**: provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
  - **Module Reviews and Takeaways**: provide on-the-job reference material to boost knowledge and skills retention.
  - **Lab Answer Keys**: provide step-by-step lab solution guidance.

Course Companion Content on the http://www.microsoft.com/learning/companionmoc Site*:* searchable, easy-to-browse digital content with integrated premium online resources that supplement the Course Handbook.

- **Modules**: include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.
- **Resources**: include well-categorized additional resources that give you immediate access to the most current premium content on TechNet, MSDN®, or Microsoft® Press®.

Student Course files on the http://www.microsoft.com/learning/companionmoc Site: includes the Allfiles.exe, a self-extracting executable file that contains all required files for the labs and demonstrations.

- ***Course evaluation:*** At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
  - To provide additional comments or feedback on the course, send an email to support@mscourseware.com. To inquire about the Microsoft Certification Program, send an email to mcphelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Microsoft® Hyper-V™ to perform the labs.

> **Important:** At the end of each lab, you must revert the virtual machines to a snapshot. You can find the instructions for this procedure at the end of each lab.

The following table shows the role of each virtual machine that is used in this course:

Virtual machine	Role
20486B-SEA-DEV11	Stand-alone machine
MSL-TMG1	Threat management gateway server

Software Configuration

The following software is installed on each virtual machine:

- Windows 8
- Visual Studio 2012 Ultimate
- Microsoft Office 2010 Professional Plus
- Microsoft Visio Premium 2010

Course Files

The files associated with the labs in this course are located in the **C:\Program Files\Microsoft Learning\20486\Drives\20486B-SEA-DEV11** folder on the student computers.

Classroom Setup

Each classroom computer will have the same virtual machine configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware is taught.

Hardware Level 7

- 64 bit Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor (2.8 Ghz dual core or better recommended)
- Dual 500 GB hard disks 7200 RPM SATA or faster (striped)
- 16 GB RAM
- DVD (dual layer recommended)
- Network adapter with Internet connectivity

- Dual SVGA monitors 17" or larger supporting 1440X900 minimum resolution
- Video adapter that supports 1440 x 900 resolution
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

In addition, the instructor computer must be connected to a projection display device that supports 1280X1024 pixels, 16 bit colors.

Module 01

Exploring ASP.NET MVC 4

Contents:

Module Overview

Microsoft ASP.NET MVC 4 and the other web technologies of the .NET Framework help you create and host dynamic, powerful, and extensible web applications. ASP.NET MVC 4 supports agile, test-driven development and the latest web standards such as HTML 5. To build robust web applications, you need to be familiar with the technologies and products in the Microsoft web stack. You also need to know how ASP.NET applications work with IIS, Visual Studio, SQL Server, Windows Azure, and Windows Azure SQL Database to deliver engaging webpages to site visitors. To choose a programming language that best suits a set of business requirements, you need to know how Model-View-Controller (MVC) applications differ from the other ASP.NET programming models: Web Pages and Web Forms.

The web application that you will design and develop in the labs throughout this course will help you develop web applications that fulfill business needs.

Objectives

After completing this module, you will be able to:

- Describe the role of ASP.NET in the web technologies stack, and how to use ASP.NET to build web applications.
- Describe the three programming models available in ASP.NET: Web Pages, Web Forms, and MVC—and select an appropriate model for a given project.
- Distinguish between an MVC Model, an MVC Controller, and an MVC View.

Lesson 1
Overview of Microsoft Web Technologies

Before you use ASP.NET MVC 4, you need to know where Microsoft ASP.NET 4.5 fits in the context of the entire Microsoft web technologies stack. You should know how ASP.NET 4.5 websites are hosted in Internet Information Server 8 (IIS) and Windows Azure, and how they run server-side code on web servers, and client-side code on web browsers, to help provide rich and compelling content.

Lesson Objectives

After completing this lesson, you will be able to:

- Provide an overview of Microsoft web technologies.
- Provide an overview of ASP.NET 4.5.
- Provide an overview of client-side web technologies, including AJAX and JavaScript libraries.
- Describe the role of IIS in the Microsoft web technologies stack.
- Describe the role of Windows Azure in the Microsoft web technologies stack.

Introduction to Microsoft Web Technologies

Microsoft provides a broad range of technologies that you can use to create rich web applications and publish them on intranets and over the Internet. Besides publishing web applications, you can use these technologies to develop and host websites, which run code both on the web server and on the user's web browser.

Develop	Host	Execute	
		Server-Side	Client-Side
• WebMatrix • Visual Studio • Visual Studio Express	• IIS • SQL Server • Windows Azure • SQL Database	• ASP.NET	• JavaScript • jQuery • AJAX

Developer Tools

You can create simple websites with text and images by using a text editor, such as Notepad. However, most websites require complex actions to be performed on the server-side, such as database operations, email delivery, complex calculations, or graphics rendering. To create such complex, highly functional, and engaging websites quickly and easily, Microsoft provides the following tools:

- *WebMatrix 2*. You can use WebMatrix 2 to create static HTML pages and dynamic pages with ASP.NET, PHP, and Node.js. WebMatrix 2 is a free development tool that you can install by downloading and using the Microsoft Web Platform Installer (Web PI) from the Microsoft website. WebMatrix 2 enables you to develop custom websites based on popular web applications such as Orchard, Umbraco CMS, and WordPress. Using WebMatrix 2, you can create ASP.NET Web Pages applications, but not ASP.NET Web Forms or MVC applications.
- *Microsoft Visual Studio 2012*. You can use Visual Studio 2012, an Integrated Development Environment (IDE), to create custom applications based on Microsoft technologies, regardless of whether these applications run on the web, on desktops, on mobile devices, or by using Microsoft cloud services. Visual Studio 2012 has rich facilities for designing, coding, and debugging any ASP.NET web application, including MVC applications.
- *Microsoft Visual Studio Express 2012 for Web*. You can use Visual Studio Express 2012 for Web to create Web Forms or MVC web applications. This is a free tool that does not include all the

features of Visual Studio 2012 editions. However, you can use it to create fully functional MVC websites.

Hosting Technologies

Regardless of the tool you use to build a web application, you must use a web server to host the web application. When users visit your site, the host server responds by rendering HTML and returning it to the user's browser for display. The host server may query a database before it builds the HTML, and the host server may perform other actions such as sending emails or saving uploaded files. You need to test the functionality of such user actions on a web server. When you build a web application by using Visual Studio 2012, you can use Visual Studio Development Server, the built-in web server to run the application. However, Visual Studio Development Server cannot host deployed web applications. Therefore, when you finish building the web application and make it ready for users to access on an intranet or over the Internet, you must use a fully functional web server such as:

- *Microsoft Internet Information Server 8*. IIS is an advanced website hosting technology. You can install IIS servers on your local network or perimeter network, or employ IIS servers hosted by an Internet service provider (ISP). IIS can host any ASP.NET, PHP, or Node.js websites. You can scale up IIS to host large and busy websites by configuring server farms that contain multiple IIS servers, all serving the same content.
- *Windows Azure*. Windows Azure is a cloud platform that provides on-demand services to build, deploy, host, and manage web applications through Microsoft-managed data centers. When you use Windows Azure services, you need to pay only for the data that your website serves. Also, you need not worry about building a scalable infrastructure because Windows Azure automatically adds resources as your website grows.

 **Note:** There are many other non-Microsoft technologies that you can use to host websites. The most popular, for example, is Apache. Apache is most frequently run on Linux servers and it is often paired with MySQL to manage databases and PHP as a server-based web framework. This configuration is often referred to as the Linux Apache MySQL PHP (LAMP) web stack.

 **Note:** Windows Azure and IIS can host websites written in ASP.NET, PHP, or Node.js. Apache web servers can be used to host a number of server-side web technologies, including PHP, Node.js, and Python, but Apache web servers cannot be used to host ASP.NET websites.

Most websites require a database to manage data such as product details, user information, and discussion topics. You can choose from the following Microsoft technologies to manage your data:

- *Microsoft SQL Server 2012*. SQL Server 2012 is a premium database server that you can use to host any database from the simplest to the most complex. SQL Server can scale up to support very large databases and very large numbers of users. You can build large SQL Server clusters to ensure the best availability and reliability. Many of the world's largest organizations rely on SQL Server to host data.
- *Windows Azure SQL Database*. SQL Database is a cloud database platform and a part of Windows Azure. Using SQL Database, you can deploy your database and pay only for the data that you use. You need not worry about managing your database infrastructure because your database scales up automatically as your website grows.

Microsoft provides a broad range of technologies that you can use to create rich web applications and publish them on intranets and over the Internet. Besides publishing web applications, you can use these technologies to develop and host websites, which run code both on the web server and on the user's web browser.

Developer Tools

You can create simple websites with text and images by using a text editor, such as Notepad. However, most websites require complex actions to be performed on the server-side, such as database operations, email delivery, complex calculations, or graphics rendering. To create such complex, highly functional, and engaging websites quickly and easily, Microsoft provides the following tools:

- *WebMatrix 2*. You can use WebMatrix 2 to create static HTML pages and dynamic pages with ASP.NET, PHP, and Node.js. WebMatrix 2 is a free development tool that you can install by downloading and using the Microsoft Web Platform Installer (Web PI) from the Microsoft website. WebMatrix 2 enables you to develop custom websites based on popular web applications such as Orchard, Umbraco CMS, and WordPress. Using WebMatrix 2, you can create ASP.NET Web Pages applications, but not ASP.NET Web Forms or MVC applications.
- *Microsoft Visual Studio 2012*. You can use Visual Studio 2012, an Integrated Development Environment (IDE), to create custom applications based on Microsoft technologies, regardless of whether these applications run on the web, on desktops, on mobile devices, or by using Microsoft cloud services. Visual Studio 2012 has rich facilities for designing, coding, and debugging any ASP.NET web application, including MVC applications.
- *Microsoft Visual Studio Express 2012 for Web*. You can use Visual Studio Express 2012 for Web to create Web Forms or MVC web applications. This is a free tool that does not include all the features of Visual Studio 2012 editions. However, you can use it to create fully functional MVC websites.

Hosting Technologies

Regardless of the tool you use to build a web application, you must use a web server to host the web application. When users visit your site, the host server responds by rendering HTML and returning it to the user's browser for display. The host server may query a database before it builds the HTML, and the host server may perform other actions such as sending emails or saving uploaded files. You need to test the functionality of such user actions on a web server. When you build a web application by using Visual Studio 2012, you can use Visual Studio Development Server, the built-in web server to run the application. However, Visual Studio Development Server cannot host deployed web applications. Therefore, when you finish building the web application and make it ready for users to access on an intranet or over the Internet, you must use a fully functional web server such as:

- *Microsoft Internet Information Server 8*. IIS is an advanced website hosting technology. You can install IIS servers on your local network or perimeter network, or employ IIS servers hosted by an Internet service provider (ISP). IIS can host any ASP.NET, PHP, or Node.js websites. You can scale up IIS to host large and busy websites by configuring server farms that contain multiple IIS servers, all serving the same content.
- *Windows Azure*. Windows Azure is a cloud platform that provides on-demand services to build, deploy, host, and manage web applications through Microsoft-managed data centers. When you use Windows Azure services, you need to pay only for the data that your website serves. Also, you need not worry about building a scalable infrastructure because Windows Azure automatically adds resources as your website grows.

 **Note:** There are many other non-Microsoft technologies that you can use to host websites. The most popular, for example, is Apache. Apache is most frequently run on Linux servers and it is often paired with MySQL to manage databases and PHP as a server-based web framework. This configuration is often referred to as the Linux Apache MySQL PHP (LAMP) web stack.

 **Note:** Windows Azure and IIS can host websites written in ASP.NET, PHP, or Node.js. Apache web servers can be used to host a number of server-side web technologies, including PHP, Node.js, and Python, but Apache web servers cannot be used to host ASP.NET websites.

Most websites require a database to manage data such as product details, user information, and discussion topics. You can choose from the following Microsoft technologies to manage your data:

- *Microsoft SQL Server 2012*. SQL Server 2012 is a premium database server that you can use to host any database from the simplest to the most complex. SQL Server can scale up to support very large databases and very large numbers of users. You can build large SQL Server clusters to ensure the best availability and reliability. Many of the world's largest organizations rely on SQL Server to host data.
- *Windows Azure SQL Database*. SQL Database is a cloud database platform and a part of Windows Azure. Using SQL Database, you can deploy your database and pay only for the data that you use. You need not worry about managing your database infrastructure because your database scales up automatically as your website grows.

Code Execution Technologies

The code that you write in a developer tool must run in one of two locations:

- *On the web server*. This code has full access to the power of the web server and any databases attached to it. It can access the database quickly, send email messages, and render webpages.
- *On the user's web browser*. This code responds quickly to user actions, such as mouse clicks, but it is more limited in what it can accomplish without interacting with the web server.

You need to use different technologies to run server-side code and client-side code.

Server-Side Execution

Microsoft ASP.NET 4.5 is a server-side web environment that runs server-side .NET code that you write in Visual Studio 2012 or WebMatrix 2. The code accesses the database, renders HTML pages, and returns them to the web browser. The MVC programming model is a part of ASP.NET 4.5. Other server-side technologies include PHP and Node.js.

Client-Side Execution

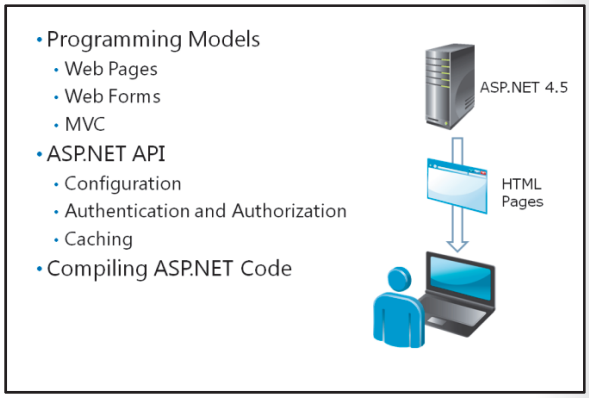
Most web browsers can run code written in the JavaScript language. This code is sent to the browser as text within a rendered HTML page or in a separate .js file. Because JavaScript is local to the browser, it can respond very quickly to user actions such as clicking, pointing, or dragging.

Many JavaScript libraries are available to accelerate client code development. For example, the popular jQuery library makes it simple to access page elements and manipulate them by changing their style or content. When you use such libraries, you can write functions in a few lines that would otherwise require hundreds of lines of your own JavaScript code.

ASP.NET applications can also use the Asynchronous JavaScript and XML (AJAX) technology on the client computer. You can use AJAX to update a small section of an HTML page, instead of reloading the entire page from the server. Such partial page updates help make webpages more responsive and engaging.

Question: If you want to animate a page element, for example, by fading it in, would you write server-side or client-side code?

Overview of ASP.NET 4.5



You can use ASP.NET 4.5 to develop database-driven, highly-functional, and scalable dynamic websites that use client-side and server-side code. You can create many kinds of website with ASP.NET 4.5, for example, web portals, online shopping sites, blogs, and wikis.

Programming Models

When you use ASP.NET 4.5 to build an application, you are not restricted to a single style of programming; instead, you can choose from three different programming models. Each programming model has a typical structure in the development environment and stores code in different places in the web hierarchy:

- Web Pages: When you build a site by using Web Pages, you can write code by using the C# or Visual Basic programming language. If you write C# code, these pages have a .cshtml file extension. If you write Visual Basic code, these pages have a .vbhtml file extension. ASP.NET runs the code in these pages on the server to render data from a database, respond to a form post, or perform other actions. This programming model is simple and easy to learn, and is suited for simple data-driven sites. ASP.NET 4.5 includes Web Pages version 2.0.
- Web Forms: When you build a site by using Web Forms, you employ a programming model with rich server-side controls and a page life cycle that is not unlike building desktop applications. Built-in controls include buttons, text boxes, and grid views for displaying tabulated data. You can also add third-party controls or build custom controls. To respond to user actions, you can attach event handlers containing code to the server-side controls. For example, to respond to a click on a button called **firstButton**, you could write code in the **firstButton_Click()** event handler.
- MVC: When you build a site by using ASP.NET MVC, you separate server-side code into three parts:
  - *Model*. An MVC model defines a set of classes that represent the object types that the web application manages. For example, the model for an ecommerce site might include a Product model class that defines properties such as Description, Catalog Number, Price, and others. Models often include data access logic that reads data from a database, and writes data to that database.
  - *Controllers*. An MVC controller is a class that handles user interaction, creates and modifies model classes, and selects appropriate views. For example, when a user requests full details about a particular product, the controller creates a new instance of the Product model class and passes it to the Details view, which displays it to the user.
  - *Views*. An MVC view is a component that builds the webpages that make up the web application's user interface. Controllers often pass an instance of a model class to a view. The view displays properties of the model class. For example, if the controller passes a Product object, the view might display the name of the product, a picture, and the price.

This separation of model, view, and controller code ensures that MVC applications have a logical structure, even for the most complex sites. It also improves the testability of the application. Ultimately, ASP.NET MVC enables more control over the generated HTML than either Web Pages or Web Forms.

 **Additional Reading:** In Lesson 2, Understanding ASP.NET 4.5, you will see how to choose the most appropriate programming model for a given website project.

The ASP.NET API

Whichever programming model you choose, you have access to classes from the ASP.NET API. These classes are included in the .NET Framework in namespaces within the **System.Web** namespace and can be used to rapidly implement common website functionalities such as:

- *Configuration*. Using Web.config files, you can configure your web application, regardless of the programming model. Web.config files are XML files with specific tags and attributes that the ASP.NET 4.5 runtime accepts. For example, you can configure database connections and custom error pages in the Web.config file. In code, you can access the configuration through the **System.Web.Configuration** namespace.
- *Authentication and Authorization*. Many websites require users to log on by entering a user name and password, or by providing extra information. You can use ASP.NET membership providers to authenticate and authorize users, and restrict access to content. You can also build pages that enable users to register a new account, reset a password, recover a lost password, or perform other account management tasks. Membership providers belong to the **System.Web.Security** namespace.
- *Caching*. ASP.NET may take some time to render a complex webpage that may require multiple database queries or calls to external web services. You can use caching to mitigate this delay. ASP.NET caches a rendered page in memory, so that it can return the same page to subsequent user requests without having to render it again from the start. In a similar manner, .NET Framework objects can also be cached. You can access cached pages by using the **System.Runtime.Caching** namespace and configure the caches in Web.config.

Compiling ASP.NET Code

Because ASP.NET server-side code uses the .NET Framework, you must write code in a .NET managed programming language such as C# or Visual Basic. Before running the code, it must be compiled into native code so that the server CPU can process it. This is a two-stage process:

1. *Compilation to MSIL*. When you build a website in Visual Studio, the ASP.NET compiler creates .dll files with all the code compiled into Microsoft Intermediate Language (MSIL). This code is both independent of the language you used to write the application and the CPU architecture of the server.
2. *Compilation to native code*. When a page is requested for the first time, the Common Language Runtime (CLR) compiles MSIL into native code for the server CPU.

This two-stage compilation process enables components written in different languages to work together and enables many errors to be detected at build time. Note, however, that pages may take extra time to render the first time they are requested after a server restart. To avoid this delay, you can pre-compile the website.

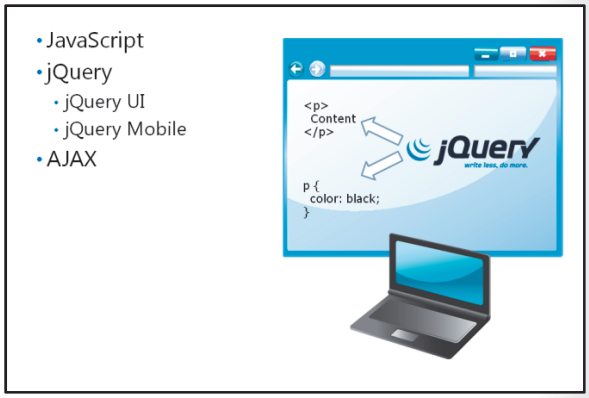
When you use the default compilation model, delays can arise when the first user requests a page. This is because ASP.NET must compile the page before serving it to the browser. To avoid such delays and to protect source code, use pre-compilation. When you pre-compile a site, all the ASP.NET files, including controllers, views, and models, are compiled into a single .dll file.

 **Additional Reading:** For more information about ASP.NET compilation, see: http://go.microsoft.com/fwlink/?LinkID=293680&clcid=0x409

Question: Which of the three programming models do you think provides the most control over the HTML and JavaScript code that is sent to the browser?

Client-Side Web Technologies

Originally, in ASP.NET, and similar technologies like PHP, all the code ran on the web server. This approach is often practical because the web server usually has immediate access to the database and more processor power and memory than a typical client computer. However, in such an approach, every user action requires a round trip between the client and the web server, and most actions require a complete reload of the page. This can take significant time. To respond quickly and provide better user experience, you can supplement server-side code with client-side code that runs in the web browser.



JavaScript

JavaScript is a simple scripting language that has syntax similar to C#, and it is supported by most web browsers. A scripting language is not compiled. Instead, a script engine interprets the code at run time so that the web browser can run the code.

Note: Besides JavaScript, Internet Explorer supports VBScript. There are other scripting languages also, but JavaScript is supported by virtually every web browser. This is not true of any other scripting language. Unless your target audience is very limited and you have control over the browser used by your users, you should use JavaScript because of its almost universal support.

You can include JavaScript code in your ASP.NET pages, irrespective of the programming model you choose. JavaScript is a powerful language, but can require many lines of code to achieve visual effects or call external services. Script libraries contain pre-built JavaScript functions that help implement common actions that you might want to perform on client-side code. You can use a script library, instead of building all your own JavaScript code from the start; using a script library helps reduce development time and effort.

Different browsers interpret JavaScript differently. When you develop an Internet site, you do not know what browsers site visitors use. Therefore, you must write JavaScript that works around browser compatibility.

jQuery

jQuery is one of the most popular JavaScript libraries. It provides elegant functions for interacting with the HTML elements on your page and with cascading style sheet (CSS) styles. For example, you can locate all the ** ** elements on a webpage and change their background color by using a single line of code. To achieve the same result by using JavaScript only, you need to write several lines of code and a programming loop. Furthermore, the code you write may be different for different browsers. Using jQuery, it is easier to write code to respond to user actions and to create simple animations. jQuery also handles browser differences. You can use jQuery to call web services on remote computers and update the webpage with the results returned.

The jQuery project also includes two other JavaScript libraries that extend the base jQuery library:

- *jQuery UI*. This library includes a set of widgets that you can use to help build a user interface. There are date pickers, spinners, dialog boxes, autocomplete text boxes, and other widgets. You can also apply themes to your jQuery UI widgets to integrate their colors and styles with the website branding.
- jQuery *Mobile*. This library makes it easy to provide a user interface for mobile devices such as phones and tablets. It renders HTML by using progressive enhancement. For example, using jQuery Mobile, you can display rich, advanced controls on advanced mobile devices such as smartphones with large screens, whereas you can display a simpler UI with the same functionality for older mobile devices with smaller screens.

Note: There are many other JavaScript libraries such as Prototype, Enyo, Ext, and Dojo Toolkit. If you find any of these better suited to a particular task, or if you have experience in developing web applications by using them, you can include them in your ASP.NET pages, instead of jQuery.

AJAX

AJAX is a technology that enables browsers to communicate with web servers asynchronously by using the XmlHttpRequest object without completely refreshing the page. You can use AJAX in a page to update a portion of the page with new data, without reloading the entire page. For example, you can use AJAX to obtain the latest comments on a product, without refreshing the entire product page.

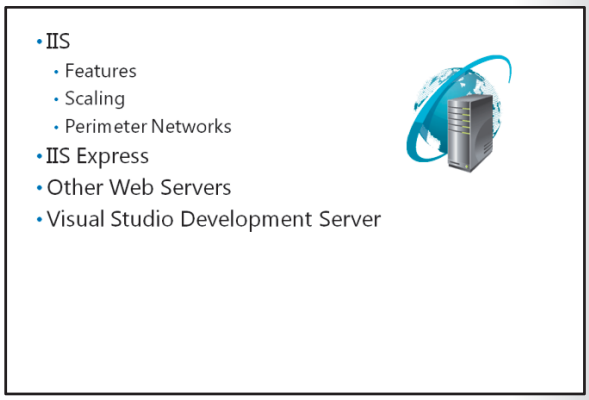
AJAX is an abbreviation of Asynchronous JavaScript and XML. AJAX is implemented entirely in JavaScript, and ASP.NET 4.5, by default, relies on the jQuery library to manage AJAX requests and responses. The code is run asynchronously, which means that the web browser does not freeze while it waits for an AJAX response from the server. Initially, developers used XML to format the data returned from the server. More recently, however, developers use JavaScript Object Notation (JSON) as an alternative format to XML.

Question: Can you think of any problems that might arise if you include the jQuery library with every page in your application?

Internet Information Server 8.0

Every website must be hosted by a web server. A web server receives requests for web content from browsers, runs any server-side code, and returns webpages, images, and other content. HTTP is used for communication between the web browser and the web server.

Internet Information Server 8.0 is a web server that can scale up from a small website running on a single web server to a large website running on a multiple web server farms. Internet Information Server 8.0 is available with Windows Server 2012.



Internet Information Server 8.0 Features

Internet Information Server 8.0 is tightly integrated with ASP.NET 4.5, Visual Studio 2012, and Windows Server 2012. It includes the following features:

- *Deployment Protocols*. The advanced Web Deploy protocol, which is built into Visual Studio 2012, automatically manages the deployment of a website with all its dependencies. Alternatively, you can use File Transfer Protocol (FTP) to deploy content.
- *Centralized Web Farm Management*. When you run a large website, you can configure a load-balanced farm of many IIS servers to scale to large sizes. IIS management tools make it easy to deploy sites to all servers in the farm and manage sites after deployment.
- *High Performance Caches*. You can configure ASP.NET to make optimal use of the IIS caches to accelerate responses to user requests. When IIS serves a page or other content, it can cache it in memory so that subsequent identical requests can be served faster.
- *Authentication and Security*. IIS supports most common standards for authentication, including Smart Card authentication and Integrated Windows authentication. You can also use Secure Sockets Layer (SSL) to encrypt security-sensitive communications, such as logon pages and pages containing credit card numbers.
- *ASP.NET Support*. IIS is the only web server that fully supports ASP.NET.
- *Other Server-Side Technologies*. You can host websites developed in PHP and Node.js on IIS.

Scaling Up IIS

A single web server has limited scalability because it is limited by its processor speed, memory, disk speed, and other factors. Furthermore, single web servers are vulnerable to hardware failures and outages. For example, when a single web server is offline, your website is unavailable to users.

You can improve the scalability and resilience of your website by hosting it on a multiple server farm. In such server farms, many servers share the same server name. Therefore, all servers can respond to browser requests. A load balancing system such as Windows Network Load Balancing or a hardware-based system such as Riverbed Cascade, distributes requests evenly among the servers in the server farm. If a server fails, other servers are available to respond to requests, and thereby, the website availability is not interrupted. IIS servers are designed to work in such server farms and include advanced management tools for deploying sites and managing member servers.

Perimeter Networks

Web servers, including IIS, are often located on a perimeter network. A perimeter network has a network segment that is protected from the Internet through a firewall that validates and permits incoming HTTP requests. A second firewall, which permits requests only from the web server, separates the perimeter network from the internal organizational network. Supporting servers, such as database servers, are usually located on the internal organizational network.

In this configuration, Internet users can reach the IIS server to request pages and the IIS server can reach the database server to run queries. However, Internet users cannot access the database server or any other internal computer directly. This prevents malicious users from running attacks and ensures a high level of security.

IIS Express

Internet Information Server 8.0 Express does not provide all the features of Internet Information Server 8.0 on Windows Server 2012. For example, you cannot create load-balanced server farms by using Internet Information Server 8.0 Express. However, it has all the features necessary to host rich ASP.NET 4.5 websites and other websites on a single server. Internet Information Server 8.0 Express is included with Windows 8. You can also install it on Windows Server 2012, Windows Server 2008, Windows 7, and Windows Vista SP1 and later by downloading and using the Web Platform Installer (Web PI).

Other Web Servers

Apache is a popular non-Microsoft web server and there are other alternatives such as nginx. Apache can be installed on Windows Server or Windows client computers to host websites during development or for production deployments. However, Apache cannot host ASP.NET websites.

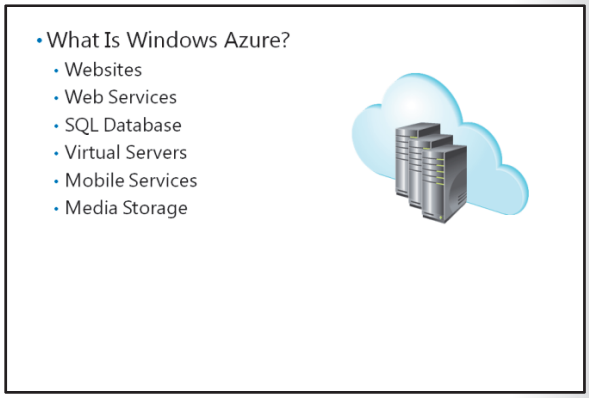
Visual Studio Development Server

Visual Studio 2012 includes a built-in web server, Visual Studio Development Server. When you open a website in debugging mode, Visual Studio Development Server starts and hosts your site. If you close the browser or stop debugging, Visual Studio Development Server automatically stops. Visual Studio Development Server cannot host sites for production and does not provide all the functionality available in Internet Information Server 8.0 Express. For example, you cannot configure SSL encryption in Visual Studio Development Server.

Question: If you wanted to host an ASP.NET site you had written for simple testing by a small team of developers, which of the preceding web servers would you use as a host?

Windows Azure

Windows Azure is a part of Microsoft cloud services for hosting business-critical systems. When you run code on Windows Azure, the code runs on servers at Microsoft-managed data centers at locations around the globe. You have two main advantages when you use Windows Azure to host and deploy your web application:



- Flexible Scaling: As the needs of your web application and database grow, extra server resources are automatically allocated. You do not need to plan for server farms or load balancing systems because these are built into Windows Azure.
- Flexible Pricing: With Windows Azure, you can choose a pay-as-you-go pricing model, which means that you only pay for the data that you use. This makes Windows Azure very cost-efficient for small websites. It also makes costs predictable for large websites.

You can use Windows Azure to host the following:

- *Websites*. You can host an entire website on Windows Azure. Windows Azure supports websites developed in ASP.NET, PHP, or Node.js. You can also deploy websites to Windows Azure directly from Visual Studio 2012 or WebMatrix 2.
- *Web services*. A web service does not include a user interface like a website. Instead, it consists of a set of methods that you can call, from other software. You can call web services from mobile devices, desktop applications, and from website code running on the client-side or the server-side.
- *Databases*. When you use Internet Information Server 8.0 to host a website, you can use SQL Server to store the website database. When you host a website in Windows Azure, you can use SQL Database, which is a cloud-based database service based on SQL Server, to host your database.

- *Virtual servers*. You can provision entire virtual servers in Windows Azure to run business-critical back office software or use the virtual servers as test environments. Virtual servers in Windows Azure can run Windows Server or Linux.
- *Mobile services*. If you develop apps for mobile devices such as phones and tablets, you can use Windows Azure to host services that underpin them. Such services can provide user preference storage, data storage and other functions.
- *Media services*. When you use Windows Azure to host media such as audio and video, it is automatically available to many types of devices such as computers, mobile phones, and tablets, and it is encoded in various formats, such as MP4 and Windows Media formats.

Question: How would site visitors know if your site is hosted in Windows Azure or IIS Server?

Lesson 2
Overview of ASP.NET 4.5

ASP.NET 4.5 helps you create dynamic websites that use client-side and server-side code. In addition, with ASP.NET 4.5, you are not restricted to a single style of programming; instead, you can choose from three different programming models: Web Pages, Web Forms, and MVC. These programming models differ from each other, and they have their own advantages and disadvantages in different scenarios. ASP.NET also provides many features that you can use, regardless of the programming model you choose.

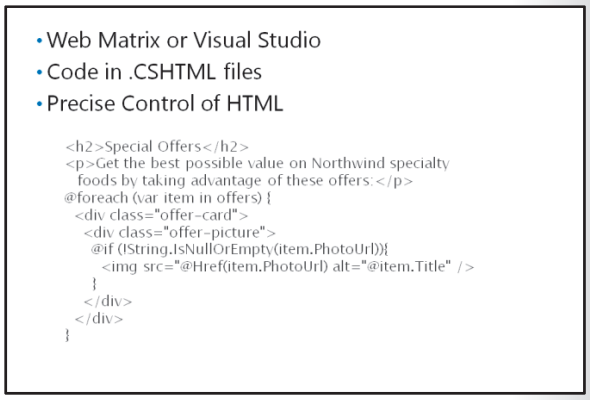
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Web Pages programming model.
- Describe the Web Forms programming model.
- Describe the MVC programming model.
- Determine whether to build Web Pages, Web Forms, or MVC web applications, based on customer needs.
- Describe the features that can be used in all ASP.NET applications, regardless of the chosen programming model.

Web Pages Applications

Web Pages is the simplest programming model you can choose to build a web application in ASP.NET. You can use Web Pages to create a website quickly and with little technical knowledge. There is a single file associated with each webpage that users can request. For each page, you write HTML, client-side code, and server-side code in the same .cshtml file. The @ symbol is used to distinguish server-side code from HTML and JavaScript. When users request the page, the ASP.NET runtime compiles and runs the server-side code to render HTML and returns that HTML to the web browser for display.



Note: If you want to write server-side code in Visual Basic, you use .vbhtml files, instead of .cshtml files.

You can use either Visual Studio 2012 or WebMatrix 2 to create Web Pages applications. ASP.NET 4.5 includes version 2.0 of Web Pages.

The following code example shows HTML and C# code in part of a Web Pages file. The code displays data from the item object, which is an MVC model class.

A Web Pages Code Example

```
<h2>Special Offers</h2>
<p>Get the best possible value on Northwind specialty foods by taking advantage of these
offers:</p>
```

```
@foreach (var item in offers) {
<div class="offer-card">
<div class="offer-picture">
      @if (!String.IsNullOrEmpty(item.PhotoUrl)){
<img src="@Href(item.PhotoUrl) alt="@item.Title" />
      }
</div>
</div>
}
```

The Web Pages programming model has the following advantages:

- It is simple to learn.
- It provides precise control over the rendered HTML.
- It can be written in WebMatrix 2 or Visual Studio 2012.

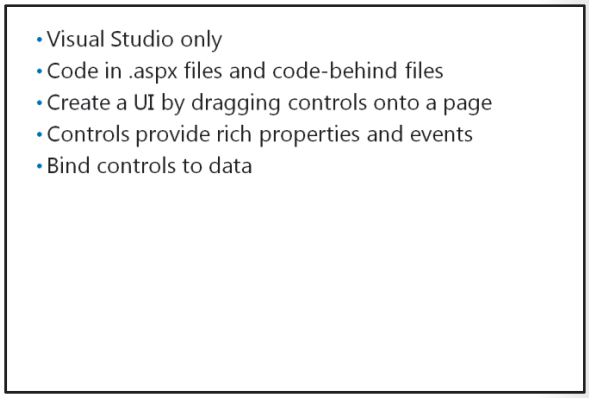
Using a Web Pages site has some disadvantages:

- It provides no control over URLs that appear in the Address bar.
- Large websites require a large numbers of pages, each of which must be coded individually.
- There is no separation of business logic, input logic, and the user interface.

Question: Why should web developers need to take control of the URLs that appear in the Address bar when a visitor browses a Web Pages site?

Web Forms Applications

Web Forms is another programming model that you can choose in ASP.NET. WebMatrix 2 does not support Web Forms, so you must use Visual Studio 2012 to develop a Web Forms application. A Web Forms application is characterized by controls, which you can drag from the Visual Studio toolbox onto each webpage. This method of creating a user interface resembles the method used in desktop applications.



Web Forms Controls

ASP.NET provides a wide variety of highly-functional controls that you assemble on Web Forms. After you add a control to a page, you can write code to respond to user events. For example, you can use code in a button click event to process a user's input in a form. The controls provided include:

- Input controls, such as text boxes, option buttons, and check boxes.
- Display controls, such as image boxes, image maps, and ad rotators.
- Data display controls, such as grid views, form views, and charts.
- Validation controls, which check data entered by the user.
- Navigation controls, such as menus and tree views.

You can also create your own custom controls to encapsulate custom functionality.

Web Forms Code Files

In a Web Forms application, HTML and control markup is stored in files with an .aspx extension. Server-side C# code is usually written in an associated .cs file called a code-behind file. For example, a page called Default.aspx usually has a code-behind file called Default.aspx.cs.

Similarly, when you write custom controls, you store HTML and control markup in an .ascx file. A control called CustomControl.ascx has a code-behind file called CustomControl.ascx.cs.

Web Forms applications can also contain class files that have the .cs extension.

 **Note:** If you write server-side code in Visual Basic, code-behind files have a .vb extension, instead of a .cs extension.

Binding Controls to Data

In Web Forms applications, you can easily display data by binding controls to data sources. This technique removes the necessity to loop through data rows and build displays line-by-line. For example, to bind a grid view control to a SQL Server database table, you drag a SQL data source control onto the Web Form, and use a dialog to bind the grid view to the data source. When the page is requested, ASP.NET runs the query on the data source and merges the returned rows of data with the Web Forms page.

Advantages and Disadvantages of Web Forms

The Web Forms programming model has the following advantages:

- You can design your page visually by using server controls and Design View.
- You can use a broad range of highly functional controls that encapsulate a lot of functionality.
- You can display data without writing many lines of server-side code.
- The user interface in the .aspx file is separated from input and business logic in the code-behind files.

Using a Web Forms site has some disadvantages:

- The ASP.NET Web Forms page life cycle is an abstraction layer over HTTP and can behave in unexpected ways. You must have a complete understanding of this life cycle, to write code in the correct event handlers.
- You do not have precise control over markup generated by server-side controls.
- Controls can add large amounts of markup and state information to the rendered HTML page. This increases the time taken to load pages.

Question: Why should web developers be concerned about the markup and state information that ASP.NET Web Forms controls add to a rendered HTML page?

MVC Applications

MVC is another programming model available in ASP.NET. MVC applications are characterized by a strong separation of business logic, data access code, and the user interface into Models, Controllers, and Views. ASP.NET 4.5 includes MVC version 4.0.

You cannot use WebMatrix to create MVC applications.

- Visual Studio only
- Models encapsulate objects and data
- Views generate the user interface
- Controllers interact with user actions
- Code in .cshtml and .cs files
- Precise control of HTML and URLs
- Easy to use unit tests

Models

Each website presents information about different kinds of objects to site visitors. For example, a publisher's website may present information about books and authors. A book includes properties such as the title, a summary, and the number of pages. An author may have properties such as a first name, a last name, and a short biography. Each book is linked to one or more authors.

When you write an MVC website for a publisher, you would create a model with a class for books and a class for authors. These model classes would include the properties described and may include methods such as "buy this book" or "contact this author". If books and authors are stored in a database, the model can include data access code that can read and change records.

Models are custom .NET classes and store code in .cs files.

Views

Each website must render HTML pages that a browser can display. This rendering is completed by Views. For example, in the publishing site, a View may retrieve data from the Book Model and render it on a webpage so that the user can see the full details. In MVC applications, Views create the user interface.

Views are markup pages that store both HTML and C# code in .cshtml files. This means that they are like Web Pages, but they include only user interface code. Other logic is separated into Models and Controllers.

Controllers

Each website must interact with users when they click buttons and links. Controllers respond to user actions, load data from a model, and pass it to a view, so that it will render a webpage. For example, in the publishing site, when the user double-clicks a book, he or she expects to see full details of that book. The Book Controller receives the user request, loads the book model with the right book ID, and passes it to the Book Details View, which renders a webpage that displays the book. Controllers implement input logic and tie Models to the right Views.

Controllers are .NET classes that inherit from the **System.Web.Mvc.Controller** class and store code in .cs files.

Advantages and Disadvantages of MVC

The MVC programming model has the following advantages:

- Views enable the developer to take precise control of the HTML that is rendered.
- You can use the Routing Engine to take precise control of URLs.
- Business logic, input logic, and user interface logic are separated into Models, Controllers, and Views.
- Unit testing techniques and Test Driven Development (TDD) are possible.

Using an MVC site has some disadvantages:

- MVC is potentially more complex to understand than Web Pages or Web Forms.
- MVC forces you to separate your concerns (models, views, and controllers). Some programmers may find this challenging.
- You cannot visually create a user interface by dragging controls onto a page.
- You must have a full understanding of HTML, CSS, and JavaScript to develop Views.

Question: When a user makes a request for a particular product in your product catalog, which component receives the request first: a model, a controller, or a view?

Discussion: ASP.NET Application Scenarios

The following scenarios describe some requirements for websites. In each case, discuss which programming model you would choose to implement the required functionality.

Which programming model will you use in the following scenarios?
- A database front-end to be hosted on an intranet
- An e-commerce site for a large software organization
- A website for a small charitable trust

Database Front-End

Your organization has its own customer relationship management system that stores data in a SQL Server database. Your team of developers wrote the user interface in Visual Studio 2012 as a desktop application. The directors now require that all computers should be able to access the application even when the desktop client application is not installed. Because all computers have a browser, you have decided to write a web application in ASP.NET to enable this.

E-Commerce Site

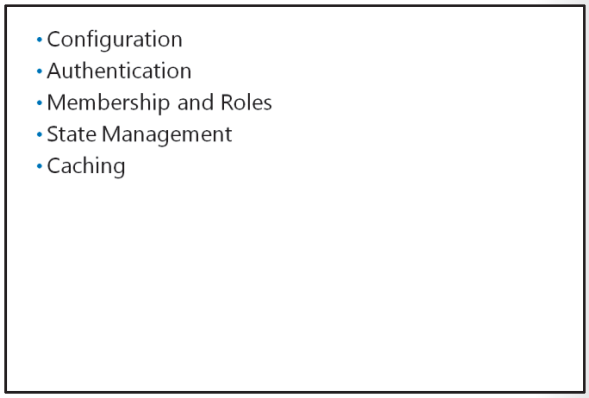
You are a consultant for a large software organization. You have been asked to architect an e-commerce website that will enable customers to browse the entire catalog of software packages, download the packages, and purchase licenses. The company has a large team of developers who are familiar with .NET object-oriented programming. The company policy is to use Test Driven Development for all software.

Website for a Small Charitable Trust

Your friend works for a charitable organization and asks your advice about a website. Your friend does not have any budget to engage a consultant, but has created websites by using Microsoft FrontPage. Your friend wants to include a database of merchandise that site visitors can browse and purchase.

Shared ASP.NET Features

ASP.NET also includes a range of features that are available regardless of the programming model that you use. This means that if you are familiar with these features from working with Web Pages or Web Forms, your knowledge can be used in MVC applications also.



Configuration

When you configure an ASP.NET site, you can control how errors are handled, how the site connects to databases, how user input is validated, and many other settings. You can configure ASP.NET sites by creating and editing Web.config files. The Web.config file in the root folder of your site configures the entire site, but you can override this configuration at lower levels by creating Web.config files in sub-folders.

Web.config files are XML files with a set of elements and attributes that the ASP.NET runtime accepts.

An Example Web.config File

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />
<add key="webpages:Version" value="2.0.0.0" />
<add key="webpages:Enabled" value="false" />
<add key="PreserveLoginUrl" value="true" />
<add key="ClientValidationEnabled" value="true" />
<add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
<connectionStrings>
<add name="ApplicationServices"
  connectionString="Data Source=
.\SQLEXPRESS;Integrated Security=SSPI;
AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
         providerName="System.Data.SqlClient"/>
</connectionStrings>
<system.web>
<customErrors mode="RemoteOnly" defaultRedirect="~/Error.html" />
<authentication mode="Forms">
<forms loginUrl="~/Account/Login" timeout="2880">
</forms>
</authentication>
<pages>
<namespaces>
<add namespace="System.Web.Helpers" />
<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.Web.WebPages" />
<add namespace="System.Web.Optimization"/>
</namespaces>
</pages>
</system.web>
</configuration>
```

If you need to access configuration values at runtime in your server-side .NET code, you can use the **System.Web.Configuration** namespace.

Authentication

Many websites identify users through authentication. This is usually done by requesting and checking credentials such as a user name and password, although authentication can be done by using more sophisticated methods, such as using smart cards. Some sites require all users to authenticate before they can access any page, but it is common to enable anonymous access to some pages and require authentication only for sensitive or subscription content.

ASP.NET supports several mechanisms for authentication. For example, if you are using Internet Explorer on a Windows computer, ASP.NET may be able to use Integrated Windows authentication. In this mechanism, your Windows user account is used to identify you. When you build Internet sites, you cannot be sure that users have Windows, a compatible browser, or an accessible account, so Forms Authentication is often used. Forms Authentication is supported by many browsers and it can be configured to check credentials against a database, directory service, or other user account stores.

Membership and Roles

In many Internet sites, for example, Facebook and Twitter, users can create their own accounts and set credentials. In this manner, your site can support a large number of members without requiring a huge amount of administrative effort because administrators do not create accounts.

In ASP.NET, a membership provider is a component that implements user account management features. Several membership providers are supported by ASP.NET, such as the SQL Membership Provider, which uses a SQL database to store user accounts. You can also create a custom membership provider, inheriting from one of the default providers, if you have unique requirements.

When you have more than a few users, you may want to group them into roles with different levels of access. For example, you might create a "Gold Members" role containing user accounts with access to the best special offers. ASP.NET role providers enable you to create and populate roles with the minimum of custom code.

You can enable access to pages on your website for individual user accounts or for all members of a role. This process is known as authorization.

State Management

Web servers and web browsers communicate through HTTP. This is a stateless protocol in which each request is separate from requests before and after it. Any values from previous requests are not automatically remembered.

However, when you build a web application, you must frequently preserve values across multiple page requests. For example, if a user places a product in a shopping cart, and then clicks "Check Out", this is a separate web request, but the server must preserve the contents of that shopping cart; otherwise, the shopping cart will be emptied and the customer will buy nothing. ASP.NET provides several locations where you can store such values or state information across multiple requests.

Caching

An ASP.NET page is built dynamically by the ASP.NET runtime on the web server. For example, in a Web Pages application, the runtime must execute the C# code in the page to render HTML to return it to the browser. That C# code may perform complex and time-consuming operations. It may run multiple queries against a database or call services on remote servers. You can mitigate these time delays by using ASP.NET caches.

For example, you can use the ASP.NET page cache to store the rendered version of a commonly requested page in the memory of the web server. The front page of your product catalog may be requested hundreds or thousands of times a day by many users. If you cache the page in memory the first time it is rendered, the web server can serve it to most users very rapidly, without querying the database server and building the page from scratch.

Question: Can you think of other facilities that all ASP.NET applications might need, regardless of the programming model they use?

Lesson 3
Introduction to ASP.NET MVC 4

You need to know how models, views, and controllers work together to render HTML, and how the structure of MVC applications determines the display of information in a Visual Studio 2012 project. You can examine the new features of MVC 4, included in ASP.NET 4.5, to build rich and engaging applications.

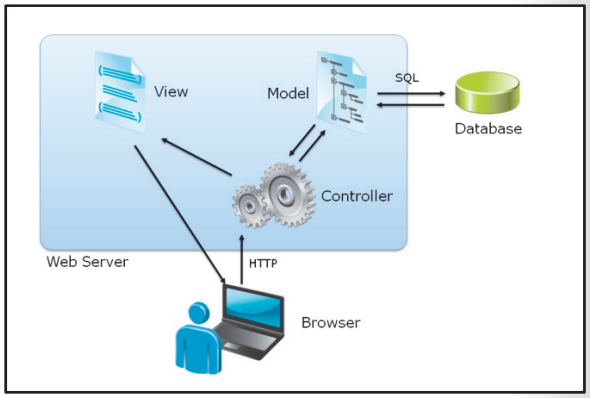
Lesson Objectives

After completing this lesson, you will be able to:

- Describe models, views, and controllers.
- Explore the main features of an ASP.NET MVC 4 web application in Visual Studio.
- Describe the new features of ASP.NET MVC 4.

Models, Views, and Controllers

Models represent data and the accompanying business logic, controllers interact with user requests and implement input logic, and views build the user interface. By examining how a user request is processed by ASP.NET MVC 4, you can understand how the data flows through models, views, and controllers before being sent back to the browser.



Models and Data

A model is a set of .NET class that represents objects handled by your website. For example, the model for an e-commerce application may include a Product model class with properties such as Product ID, Part Number, Catalog Number, Name, Description, and Price. Like any .NET class, model classes can include a constructor, which is a procedure that runs when a new instance of that class is created. You can also include other procedures, if necessary. These procedures encapsulate the business logic. For example, you can write a **Publish** procedure that marks the Product as ready-to-sell.

Most websites store information in a database. In an MVC application, the model includes code that reads and writes database records. ASP.NET MVC works with many different data access frameworks. However, the most commonly used framework is the Entity Framework version 5.0.

Controllers and Actions

A controller is a .NET class that responds to web browser requests in an MVC application. There is usually one controller class for each model class. Controllers include actions, which are methods that run in response to a user request. For example, the **Product** Controller may include a **Purchase** action that runs when the user clicks the **Add To Cart** button in your web application.

Controllers inherit from the **System.Web.Mvc.Controller** base class. Actions usually return a **System.Web.Mvc.ActionResult** object.

Views and Razor

A view is, by default, a .cshtml or .vbhtml file that includes both HTML markup and programming code. A view engine interprets view files, runs the server-side code, and renders HTML to the web browser. Razor

is the default view engine in ASP.NET MVC 4, but ASP.NET MVC 4 also supports the ASPX view engine. Additionally, you can install alternate view engines such as Spark and NHaml. The Razor view engine identifies server-side code by searching for the @ symbol, as the following code example demonstrates.

The following lines of code are part of an ASP.NET MVC 4 view and use the Razor syntax. The @ symbol delimits server-side code.

Part of a Razor View

```
<h2>Details</h2>
<fieldset>
<legend>Comment</legend>
<div class="display-label">
         @Html.DisplayNameFor(model => model.Subject)
</div>
<div class="display-field">
        @Html.DisplayFor(model => model.Subject)
</div>
<div class="display-label">
         @Html.DisplayNameFor(model => model.Body)
</div>
<div class="display-field">
        @Html.DisplayFor(model => model.Body)
</div>
</fieldset>
```

Often, the view displays properties of a model class. In the preceding code example, the **Subject** property and **Body** property are incorporated into the page.

Request Life Cycle

The Request life cycle comprises a series of events that happen when a web request is processed. The following steps illustrate the process that MVC applications follow to respond to a typical user request. The request is for the details of a product with the ID "1":

1. The user requests the web address: http://www.adventureworks.com/product/display/1
2. The MVC routing engine examines the request and determines that it should forward the request to the Product Controller and the Display action.
3. The Display action in the Product Controller creates a new instance of the Product model class.
4. The Product model class queries the database for information about the product with ID "1".
5. The Display action also creates a new instance of the Product Display View and passes the Product Model to it.
6. The Razor view engine runs the server-side code in the Product Display View to render HTML. In this case, the server-side code inserts properties such as Title, Description, Catalog Number, and Price into the HTML.
7. The completed HTML page is returned to the browser for display.

Question: If you wanted to write some code that renders data from your products catalog into an HTML table, would you place that code in a model, a view, a controller, or a JavaScript function?

Demonstration: How to Explore an MVC Application

In this demonstration, you will see how to use a sample photo sharing application to explore the structure of an MVC website.

Demonstration Steps

1. In the Solution Explorer pane of the **PhotoSharingSample – Microsoft Visual Studio** window, expand **PhotoSharingSample**, and then note that the PhotoSharingSample application does not have the default.htm, the default.aspx, or the default.cshtml files to act as a home page.
2. In the Solution Explorer pane, under PhotoSharingSample, expand **Controllers**, and then click **HomeController.cs**.
3. In the HomeController.cs code window, locate the following code.

```
Public ActionResult Index()
{
   return View();
}
```

Note: This code block represents an action that will return a view called Index.

4. In the Solution Explorer pane, expand **Views**, and then expand **Photo**.
5. In the Solution Explorer pane, under Photo, click **Index.cshtml**.
6. In the Index.cshtml code window, locate the following code.

```
<h2>@ViewBag.Title</h2>
<p>
 @Html.ActionLink("Create New", "Create")
</p>
```

Note: This code block represents the View that renders the home page.

7. On the toolbar of the **PhotoSharingSample – Microsoft Visual Studio** window, click **Internet Explorer**.
8. In the **http://localhost:**<*yourportnumber*>**/** window, note that the default home page is displayed.
9. On the taskbar, click the **Microsoft Visual Studio** icon.
10. In the **PhotoSharingSample – Microsoft Visual Studio** window, in the Solution Explorer pane, expand **App_Start**, and then click **RouteConfig.cs**.
11. In the RouteConfig.cs code window, locate the following code.

```
routes.MapRoute(
   name: "Default",
   url: "{controller}/{action}/{id}",
   )
```

Note: This code block represents the default route that forwards requests to the specified controller.

12. On the taskbar, click the **Internet Explorer** icon.
13. In the Address bar of the Windows Internet Explorer window, type the URL **http://localhost:**<*yourportnumber*>**/home/index**, and then click the **Go to** button.

Note: The browser window displays the Home page of the **http://localhost:**<*yourportnumber*>**/home/index** web application.

14. On the taskbar, click the **Microsoft Visual Studio** icon.
15. In the **PhotoSharingSample – Microsoft Visual Studio** window, in the Solution Explorer pane, expand **Models**, and then click **Photo.cs**.
16. In the Photo.cs code window, locate the following code.

```
 [Required]
public string Title { get; set;}
```

Note: This code block represents the **Title** property for a photo stored in the application.

17. In the Solution Explorer pane, under Controllers, click **PhotoController.cs**.
18. In the PhotoController.cs code window, locate the following code.

```
public class PhotoController : Controller
```

Note: This code block represents that the **PhotoController** class inherits the System.Web.MVC.Controller base class.

19. In thePhotoController.cs code window, locate the following code.

```
public ActionResult Details
    (int id = 0)
{
  Photo photo = db.Photos.Find(id);
  if (photo == null)
  {
    return HttpNotFound();
  }
  return View("Details", photo);
}
```

Note: This code block represents the **Details** action of the Photo Controller.

20. In the Solution Explorer pane, expand **Views**, expand **Photo**, and then click **Details.cshtml**.
21. In the Details.cshtml code window, locate the following code.

```
<h2>"@Model.Title"</h2>
```

Note: The Razor view engine runs this code and renders the Photo Title property that you viewed in the model.

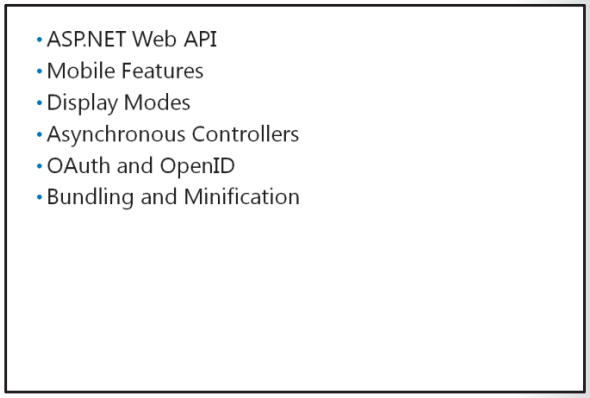
22. On the taskbar, click the **Internet Explorer** icon.
23. In the Address bar of the Windows Internet Explorer window, type **http://localhost:**<*yourportnumber*>**/photo/details/2**, and then click the **Go to** button.

Note: The photo with ID 2 is displayed in the browser window. Note that the title of the photo is rendered at the top.

24. In the Windows Internet Explorer window, click the **Close** button.
25. In the **PhotoSharingSample (Running) – Microsoft Visual Studio** window, click the **Close** button.

New Features of ASP.NET MVC 4

ASP.NET MVC 4 includes several new features, such as the ASP.NET Web API, mobile features, display modes, asynchronous controllers, **OAuth** and **OpenID** support, and bundling and minification.



- *ASP.NET Web API*. The ASP.NET Web API makes it easy to create a set of web services that can respond to browser requests by using simple HTTP verbs such as GET, POST, and DELETE. Using the Web API, you can build the back-end web services that a client-specific web application can call. Building web applications by using client-specific HTML pages and the Web API is an alternative to using ASP.NET MVC.
- *Mobile features*. As smartphones and tablets become ubiquitous, you should ensure that your website displays well on all screen sizes, resolutions, color depths, and so on. ASP.NET MVC 4.0 includes several new features to make this easier. For example, the jQuery Mobile script library renders a rich user interface for smartphone browsers, but simpler controls for older models of phones. You can use the new Mobile Project Template that uses jQuery Mobile to create sites specifically for mobile devices. Alternatively, you can add mobile-specific views to your MVC project.
- *Display modes*. Display modes enable you to easily select a view based on the web browser that made the request. Display modes help render HTML for mobile devices and they can be used to display content on unusual desktop browsers. When you add a display mode for a particular browser, you override the views, template views, and partial views that are used to render HTML.
- *Asynchronous controllers*. Usually, a controller action renders synchronously. If the action takes a long time to run, the user has to wait because the action uses a single thread. With asynchronous actions, you can call a long-running method on a separate thread and wait for the results to be returned. This enables an action to complete and respond to the user before the results are returned.
- *Support for OAuth and OpenID standards*. Using the **OAuth** and **OpenID** standards to identify users connected to the web application over the Internet, you can call the services of a third-party site to check the credentials of a user. You can trust the identity verified by the external site, and then use it to authorize access to internal resources. Using these technologies, you can, for example, enable users to access internal resources by logging on with their Windows Live account.
- *Bundling and minification*. When a browser requests a webpage, the server often returns the page with a cascading style sheet and one or more JavaScript files. ASP.NET MVC 4.0 can bundle these separate files into a single file to increase the efficiency of the response. It can also minify these

files, by removing white space, shortening variable names, and performing other actions that save space. This saves time on downloads and both server and client-side resources, and makes your web application faster.

Question: You want to encourage developers to re-use data from your website in mashups with Bing Maps. Which of the new features of MVC 4 would you use to make this possible?

Lab: Exploring ASP.NET MVC 4

Scenario

You are working as a junior developer at Adventure Works. You have been asked by a senior developer to investigate the possibility of creating a web-based photo sharing application for your organization's customers, similar to one that the senior developer has seen on the Internet. Such an application will promote a community of cyclists who use Adventure Works equipment, and the community members will be able to share their experiences. This initiative is intended to increase the popularity of Adventure Works Cycles, and thereby to increase sales. You have been asked to begin the planning of the application by examining an existing photo sharing application and evaluating its functionality. You have also been asked to examine programming models available to ASP.NET developers. To do this, you need to create basic web applications written with three different models: Web Pages, Web Forms, and MVC. Your manager has asked you to report on the following specific questions for each programming model:

- How does the developer set a connection string and data provider?
- How does the developer impose a consistent layout, with Adventure Works branding and menus, on all pages in the web application?
- How does the developer set a cascading style sheet with a consistent set of color, fonts, borders, and other styles?
- How does the developer add a new page to the application and apply the layout and styles to it?

Objectives

After completing this lab, you will be able to:

- Describe and compare the three programming models available in ASP.NET.
- Describe the structure of each web application developed in the three programming models—Web Pages, Web Forms, and MVC.
- Select an appropriate programming model for a given set of web application requirements.

Lab Setup

Estimated Time: 45 minutes

Virtual Machine: **20486B-SEA-DEV11**

User name: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

- On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.
- In the navigation pane of the **Options** dialog box, click **Package Manager**.
- Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Exploring a Photo Sharing Application

Scenario

In this exercise, you will begin by examining the photo sharing application.

The main tasks for this exercise are as follows:

1. Register a user account.
2. Upload and explore photos.
3. Use slideshows.
4. Test the authorization.

▶ Task 1: Register a user account.

1. Start the virtual machine and log on with the following credentials:
    - Virtual machine: **20486B-SEA-DEV11**
    - User name: **Admin**
    - Password: **Pa$$w0rd**
2. On the Windows 8 Start screen, open Visual Studio 2012, and pin it to the taskbar.
3. Navigate to the following location to open the **PhotoSharingSample.sln** file:
    - **Allfiles (D):\Labfiles\Mod01\PhotoSharingSample**
4. Run the web application in non-debugging mode.
5. Create a new user account with the following credentials:
    - User name: *<A user name of your choice>*
    - Password: *<A password of your choice>*

▶ Task 2: Upload and explore photos.

1. Add the following comment to the **Orchard** image:
    - Subject: **Just a Test Comment**
    - Comment: **This is a Sample**
2. Add a new photo to the application by using the following information:
    - Title of the photo: **Fall Fungi**
    - Navigation path to upload the photo: **Allfiles (D):\ Labfiles\Mod01\Pictures\fungi.jpg**
    - Description: **Sample Text**
3. Verify the description details of the newly added photo.

▶ Task 3: Use slideshows.

1. Use the **Slideshow** feature.
2. Add the following images to your list of favorite photos:
    - Fall Fungi
    - Orchard
    - Flower

3. View the slideshow of the images selected as favorites.

▶ Task 4: Test the authorization.

1. Log off from the application, and then attempt to add a comment for the Fall Fungi image.
2. Attempt to add a new photo to the Photo Index page.
3. Close the Internet Explorer window and the Visual Studio application.

Results: At the end of this exercise, you will be able to understand the functionality of a photo sharing application, and implement the required application structure in the Adventure Works photo sharing application.

Exercise 2: Exploring a Web Pages Application

Scenario

In this exercise, you will create a simple Web Pages application and explore its structure.

The main tasks for this exercise are as follows:

1. Create a Web Pages application.
2. Explore the application structure.
3. Add simple functionality.
4. Apply the site layout.

▶ Task 1: Create a Web Pages application.

1. Start Visual Studio 2012 and create a new Web Pages project by using the **ASP.NET Web Site (Razor v2)** C# template.
2. Run the new Web Pages application in Internet Explorer and review the Contact page.
3. Stop debugging by closing Internet Explorer.

▶ Task 2: Explore the application structure.

1. Open the Web.config file and verify that the database provider used by the application is **.NET Framework Data Provider for Microsoft SQL Server Compact**.
2. Verify that the Default.cshtml page and the Contact.cshtml page are linked to the same layout.
3. Verify that the Site.css file is used to apply styles to all pages on the site. Note that the _SiteLayout.cshtml page is linked to the style sheet.

▶ Task 3: Add simple functionality.

1. Add a new Razor v2 webpage to the application at the root level by using the following information:
   - Webpage name: **TestPage.cshtml**
2. Add an **H1** element to the TestPage.cshtml page by using the following information:
   - Content: **This is a Test Page**
3. Add a link to the Default.cshtml page by using the following information:
   - Start tag: **<a>**
   - Attribute: **href = "~/TestPage.cshtml"**

- Content: **Test Page**
- End tag:****

4. Save all the changes.
5. Run the website, and view the page you added.
6. Stop debugging by closing Internet Explorer.

▶ Task 4: Apply the site layout.

1. Add the Razor code block to the TestPage.cshtml file.
2. In the new code block, set the TestPage to use the following layout:
   - Layout: **_SiteLayout.cshtml**
3. Save all the changes.
4. Run the web application in debugging mode and browse to TestPage.chstml.
5. Close all open applications.

Results: At the end of this exercise, you will be able to build a simple Web Pages application in Visual Studio.

Exercise 3: Exploring a Web Forms Application

Scenario

In this exercise, you will create a simple Web Forms application and explore its structure.

The main tasks for this exercise are as follows:

1. Create a Web Forms application.
2. Explore the application structure.
3. Add simple functionality.
4. Apply the master page.

▶ Task 1: Create a Web Forms application.

1. Start Visual Studio 2012 and create a new Web Forms project, **TestWebFormsApplication**, by using the ASP.NET Web Forms Application template.
2. Run the new Web Forms application in Internet Explorer and examine the Contact page.
3. Stop debugging by closing Internet Explorer.

▶ Task 2: Explore the application structure.

1. Open the Web.config file and verify that **System.Data.SqlClient** is the database provider that the application uses.
2. Verify that the ~/Site.Master file contains a common layout for all the pages on the site. Also verify that the Default.aspx and Contact.aspx pages are linked to the same layout.
3. Verify that the Site.css file is used to apply styles to all pages on the website. Note that the Site.Master file uses bundle reference to package the CSS files.

▶ Task 3: Add simple functionality.

1. Add a new Web Forms page to the application at the route level by using the following information:
   - Name of the Web Form: **TestPage.aspx**
2. Add an **H1** element to the TestPage.aspx page by using the following information:
   - Content: **This is a Test Page**
3. Add a link to the Default.aspx page by using the following information:
   - Start tag: **<a>**
   - Attribute: **href = "TestPage.aspx"**
   - Content: **Test Page**
   - End tag:****
4. Run the website in Internet Explorer and view the newly added Web Form page.

▶ Task 4: Apply the master page.

1. Add a new attribute to the **@ Page** directive in the TestPage.aspx file by using the following information:
   - Attribute name: **MasterPageFile**
   - Attribute value: **~/Site.Master**
2. Remove the static markup tags from TestPage.aspx and replace it with a Web Forms Content control by using the following information:
   - Start tag: **<asp:Content>**
   - Runat attribute: **server**
   - ID attribute: **BodyContent**
   - ContentPlaceHolderID: **MainContent**
   - Content: **<h1>This is a Test Page</h1>**
   - End tag: **</asp:Content>**
3. Save all the changes.
4. Run the created website and verify the contents of the TestPage.aspx file.
5. Close all open applications.

Results: At the end of this exercise, you will be able to build a simple Web Forms application in Visual Studio.

Exercise 4: Exploring an MVC Application

Scenario

In this exercise, you will create a simple MVC application and explore its structure.

The main tasks for this exercise are as follows:

1. Create an MVC 4 application.

2. Explore the application structure.

3. Add simple functionality.

4. Apply the site layout.

▶ Task 1: Create an MVC 4 application.

1. Start Visual Studio 2012 and create a new **MVC** project by using the **ASP.NET MVC 4 Web Application** template. Choose the **Internet Application** template.
2. Run the new MVC application in Internet Explorer, and explore the Contact page.
3. Stop debugging by closing Internet Explorer.

▶ Task 2: Explore the application structure.

1. Open the **Web.config** file and verify whether the database provider is **System.Data.SqlClient**.
2. Verify that the ~/Views/Shared/_Layout.cshtml file contains a common layout for all pages on the website, and how pages link to the layout.
3. Verify that the Site.css file is used to apply styles to all pages on the website, and note how the pages link to the style sheet.

▶ Task 3: Add simple functionality.

1. Add a new view to the application by using the following information:
   - Parent folder: **/Views/Home**
   - Name of the view: **TestPage.cshtml**
   - Clear the **Use a layout or master page** check box.
2. Add an **H1** element to the TestPage.cshtml view by using the following information:
   - Content: **This is a Test Page**
3. Add an action to the HomeController.cs file by using the following information:
   - Procedure name: **TestPage**
   - Return type: **ActionResult**
   - Procedure parameters: **None**
   - Return the view "TestPage"
4. Add a link to the Index.cshtml page by using the following information:
   - Start tag: **<a>**
   - Attribute: **href=" ~/Home/TestPage"**
   - Content: **Test Page**
   - End tag: ****
5. Save all the changes.
6. Run the website and view the page you added.
7. Stop debugging by closing Internet Explorer.

▶ Task 4: Apply the site layout.

1. Open the TestPage.cshtml file and remove the code that sets the Layout = null.
2. In the TestPage.cshtml file, remove all the tags except the **<h1>** tag and its contents.

3. Save all the changes.
4. Run the web application and browse to Test Page.
5. Close all the open applications.

Results: At the end of this exercise, you will be able to build a simple MVC application in Visual Studio.

Question: Which of the three programming models has the simplest method of applying a single layout across multiple pages?

Question: Which of the three programming models has the simplest method of building a user interface?

Question: Which of the application programming models will you recommend for the photo sharing application: Web Pages, Web Forms, or MVC?

Module Review and Takeaways

In this module, you have seen the tools, technologies, and web servers that are available in the Microsoft web stack, which you can use to build and host web applications for intranets and on the Internet. You should also be able to distinguish applications written in the three ASP.NET programming models: Web Pages, Web Forms, and MVC. You should be able to use MVC applications to render webpages by using models, views, and controllers.

Best Practice: Use Web Pages when you have simple requirements or have developers with little experience of ASP.NET.

Best Practice: Use Web Forms when you want to create a user interface by dragging controls from a toolbox onto each webpage or when your developers have experience of Web Forms or Windows Forms.

Best Practice: Use MVC when you want the most precise control over HTML and URLs, when you want to cleanly separate business logic, user interface code, and input logic, or when you want to perform Test Driven Development.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You add a new view to an MVC application, but when you try to access the page, you receive an HTTP 404 error.	

Additional Reading: For more information about ASP.NET, including forums, blogs, and third-party tools, visit the official ASP.NET site: http://go.microsoft.com/fwlink/?LinkID=293681&clcid=0x409

Review Question(s)

Question: Which of the shared features of ASP.NET can you use in Web Pages, Web Forms, and MVC applications to increase the speed with which frequently-requested pages are returned to the browser?

Question: You want to create a simple website that shares dates and venues for games for your sports club members. The sports club has no budget to buy software. Which development environment should you use to create the site?

Real-world Issues and Scenarios

You have written a web application for a client that sells hats. Visitors to the site will be able to register, redeem offer vouchers, and purchase hats. You expect site traffic to be steady through most of the year, but to peak just before Christmas. Should you recommend IIS or Windows Azure for hosting the site?

Module 02

Designing ASP.NET MVC 4 Web Applications

Contents:

Module Overview

Microsoft ASP.NET MVC 4 is a programming model that you can use to create powerful and complex web applications. However, all complex development projects, and large projects in particular, can be challenging and intricate to fully understand. Without a complete understanding of the purposes of a project, you cannot develop an effective solution to the customer's problem. You need to know how to identify a set of business needs and plan an MVC web application to meet those needs. The project plan that you create assures stakeholders that you understand their requirements and communicates the functionality of the web application, its user interface, structure, and data storage to the developers who will create it. By writing a detailed and accurate project plan, you can ensure that the powerful features of MVC are used most effectively to solve a customer's business problems.

Objectives

After completing this module, you will be able to:

- Plan the overall architecture of an MVC 4 web application and consider aspects such as state management.
- Plan the models, controllers, and views that are required to implement a given set of functional requirements.

Lesson 1
Planning in the Project Design Phase

Before you and your team of developers plan a Model-View-Controller (MVC) web application or write any code, you must have a thorough understanding of two things: the business problem you are trying to solve and the ASP.NET components that you can use to build a solution. Before designing a web application architecture and its database, you should know how to identify the requirements that potential users of a web application have.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the various project development models.
- Describe how to gather information about project requirements when building MVC 4 web applications.
- Determine the functional requirements and business problems when building web applications.
- Explain how to plan the database design when building a web application.
- Describe possible distributed application architectures.
- Describe the options for planning state management in a web application.
- Describe the options for planning globalization and localization of a web application.
- Determine the critical aspects of web application design.

Project Development Methodologies

Developing a web application or intranet application is often a complex process that involves many developers in different teams performing various roles. To organize the development process and ensure that everybody in a project works together, you can use a wide range of development methodologies. These development methodologies describe the phases of the development project, the roles people take, the deliverables that conclude each phase, and other aspects of the project. You should choose a development methodology at an early stage in a project. Many organizations have a standard methodology that they always use for project development.

Development Model	Description
Waterfall Model	Activities for building an application are performed sequentially in distinct phases with clear deliverables.
Iterative Development Model	An application is built iteratively in parts, by using working versions that are thoroughly tested, until it is finalized.
Prototype Model	Based on a few business requirements, a prototype is made. Feedback on the prototype is used as input to develop the final application.
Agile Development Model	An application is built in rapid cycles, integrating changing circumstances and requirements in the development process.
Extreme Programming	Begins with solving a few critical tasks. Developers test the simplified solution and obtain feedback from stakeholders to derive the detailed requirements, which evolve over the project life cycle.
Test-Driven Development	A test project is created and you can test changes to the code singly or as a group, throughout the project.
Unified Modeling Language	UML diagrams are used for planning and documenting purposes, across all project development models.

Some project development methodologies include the waterfall model, the iterative development model, the prototyping model, the agile software development model, extreme programming, and test-driven development.

Waterfall Model

The waterfall model is an early methodology that defines the following phases of a project:

- *Feasibility analysis.* In this phase, planners and developers study and determine the approaches and technologies that can be used to build the software application.

- *Requirement analysis.* In this phase, planners and analysts interview the users, managers, administrators, and other stakeholders of the software application to determine their needs.
- *Application design.* In this phase, planners, analysts, and developers record a proposed solution.
- *Coding and unit testing.* In this phase, developers create the code and test the components that make up the system individually.
- *Integration and system testing.* In this phase, developers integrate the components that they have built and test the system as a whole.
- *Deployment and maintenance.* In this phase, developers and administrators deploy the solution so that users can start using the software application.

The waterfall model classifies the development project into distinct phases with a clear definition of deliverables for each phase. The model also emphasizes the importance of testing. However, the customer does not receive any functional software for review until late in the project. This makes it difficult to deal with changes to the design in response to beta feedback or manage altered circumstances.

Iterative Development Model

When you use an iterative development model, you break the project into small parts. For each part, you perform the activities related to all the stages of the waterfall model. The project is built up stage by stage, with thorough testing at each stage to ensure quality.

In an iterative project, you can perform corrective action at the end of each iteration. These corrections might reflect a better understanding of the business problems, insightful user feedback, or a better understanding of the technologies that you used to build the solution. Because requirements are added at the end of each iteration, iterative projects require a great deal of project management effort and frequently feature an overrun of planned efforts and schedule.

Prototyping Model

The prototyping model is suitable for a project where you begin with a few or meagerly defined business requirements. This situation occurs when the customers or stakeholders have only a vague understanding of their needs and how to solve them. In this approach, developers create a simplified version of the software application, and then seek feedback from stakeholders. This feedback on the prototype is used to define the detailed requirements, which developers use in the next iteration to build a solution that matches the needs of stakeholders to better help them perform their jobs.

After two or more iterations, when both stakeholders and developers reach a consensus on the requirements, a complete solution is built and tested. The prototyping model, however, can lead to a poorly-designed application because at no stage in the project is there a clear focus on the overall architecture.

Agile Software Development Model

The waterfall model, iterative development model, and prototyping model are based on the premise that business requirements and other factors do not change from the beginning to the end of the project. In reality, this assumption is often invalid. Agile software development is a methodology designed to integrate changing circumstances and requirements throughout the development process. Agile projects are characterized by:

- *Incremental development.* Software is developed in rapid cycles that build on earlier cycles. Each iteration is thoroughly tested.
- *Emphasis on people and interactions.* Developers write code based on what people do in their role, rather than what the development tools are good at.

- *Emphasis on working software*. Instead of writing detailed design documents for stakeholders, developers write solutions that stakeholders can evaluate at each iteration to validate if it solves a requirement.
- *Close collaboration with customers*. Developers discuss with customers and stakeholders on a day-to-day basis to check requirements.

Extreme Programming

Extreme programming evolved from agile software development. In extreme programming, the preliminary design phase is reduced to a minimum and developers focus on solving a few critical tasks. As soon as these critical tasks are finalized, developers test the simplified solution and obtain feedback from stakeholders. This feedback helps developers identify the detailed requirements, which evolve over the project life cycle.

Extreme programming defines a user story for every user role. A user story describes all the interactions that a user with a specific role might perform with the completed application. The collection of all the user stories for all user roles describes the entire application.

In extreme programming, developers often work in pairs. One developer writes the code and the other developer reviews the code to ensure that it uses simple solutions and adheres to best practices. Test-driven development is a core practice in extreme programming.

 **Additional Reading:** For more information about the extreme programming model, go to http://go.microsoft.com/fwlink/?LinkID=288945&clcid=0x409.

Test Driven Development

In test-driven development (TDD), developers write test code as their first task in a given iteration. For example, if you want to write a component that stores credit card details, you begin by writing tests that such a component would pass. These may be whether it checks the number formats correctly, whether it writes strings to a database table correctly, or whether it calls banking services correctly. After you define the tests, you write the component to pass those tests.

In subsequent iterations, the credit card tests remain in place. This ensures that if you break the credit card functionality, perhaps by refactoring code or by adding a new constructor, you find this out quickly because the tests fail.

In Microsoft Visual Studio 2012, you can define a test project, within the same solution as the main project, to store and run unit tests. After you write the tests, you can run them singly or in groups after every code change. Because MVC projects have the model, view, and controller code in separate files, it is easy to create unit tests for all aspects of application behavior. This is a major advantage of MVC over Web Pages and Web Forms.

Unified Modeling Language

The Unified Modeling Language (UML) is an industry standard notation to record the design of any application that uses object-oriented technology. UML is not a development model. Rather, UML diagrams are often used for planning and documenting application architecture and components, across all project development methodologies. When you use UML to design and record an application, you create a range of diagrams with standard shapes and connectors. These diagrams can be divided into three classes:

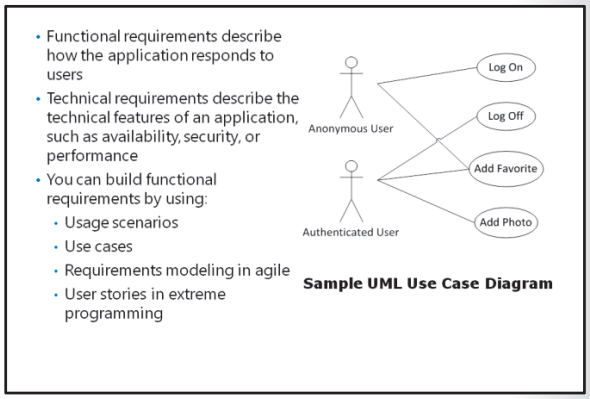
- *Behavior diagrams*. These diagrams depict the behavior of users, applications, and application components.
- *Interaction diagrams*. These diagrams are a subset of behavior diagrams that focus on the interactions between objects.

- *Structure diagrams*. These diagrams depict the elements of an application that are independent of time. This means they do not change through the lifetime of the application.

Question: What aspects of agile software development and extreme programming might be of concern to customers? What aspects might reassure them?

Gathering Requirements

When a project is commissioned, you need a vision of the solution. The vision can often be vague and require in-depth investigation before you can add details and ensure that all stakeholders' requirements are covered by the web application that you build. These requirements can be of two types:



Sample UML Use Case Diagram

- *Functional requirements*. These requirements describe how the application behaves and responds to users. Functional requirements are often called behavioral requirements. They include:
  - *User interface requirements*. These requirements describe how the user interacts with an application.
  - *Usage requirements*. These requirements describe what a user can do with the application.
  - *Business requirements*. These requirements describe how the application will fulfill business functions.
- *Technical requirements*. These requirements describe technical features of the application and relate to availability, security, or performance. These requirements are sometimes called non-functional or non-behavioral requirements.

You usually gather requirements by interviewing stakeholders such as users, administrators, other developers, board members, budget holders, and team managers. Each of these groups will have a different set of priorities that the application needs to fulfill.

Usage Scenarios and Use Cases

A common method by which you can build a set of user interface requirements, business requirements, and usage requirements is to ask users what they will do with the web application that you build. You can record these actions as usage scenarios and use cases.

A usage scenario is a specific real-world example, with names and suggested input values, of an interaction between the application and a user. The following is a simple example:

1. Roger Lengel clicks the **Add Photo** link on the main site menu.
2. Roger provides the input, **RogerL**, in the **User name** box and the password in the **Password** box to authenticate on the site.
3. Roger types the title, **Sunset**, for the photo.
4. Roger browses to the JPEG file for his new photo.
5. Roger clicks the **Upload** button.
6. The web application stores the new photo and displays the photo gallery to Roger.

A use case is similar to a usage scenario, but is more generalized. Use cases do not include user names or input values. They describe multiple paths of an interaction, which depends on what the user provides as input or other values. The following is a simple example:

1. The user clicks the **Add Photo** link on the main site menu.
2. If the user is anonymous, the logon page is shown and the user provides credentials.
3. If the credentials are correct, the **CreatePhoto** view is displayed.
4. The user types a title.
5. The user specifies the photo file to upload.
6. The user optionally types a description for the photo.
7. The user clicks the **Upload** button.
8. The web application stores the new photo and displays the photo gallery to the user.

 **Note:** Similar to verbal descriptions, you can use UML Use Case diagrams to record the use cases for your web application.

By analyzing usage scenarios and use cases, you can identify functional requirements of all types. For example, from the preceding use case, you can identify the following user interface requirement: The webpage that enables users to create a new photo must include Title and Description text boxes, a file input control for the photo file, and an Upload button to save the photo.

Agile Requirements Modeling

In a traditional waterfall model or iterative development model, developers and analysts investigate and record the precise and detailed functional and technical requirements at an early stage of the project, which do not change later. By contrast, in an agile development model-based project, developers recognize that requirements can change at any time during development. Requirements analysis is therefore characterized as follows:

- *Initial requirement modeling.* In the initial design phase, developers identify and record a few broad use cases in an informal manner without full details.
- *Just-in-time modeling.* Before writing code that implements a use case, a developer discusses it with the relevant users. At this point, the developer adds full details to the use case. In an agile development project, developers talk to users and other stakeholders at all times, and not just at the beginning and end of the project.
- *Acceptance testing.* An acceptance test is a test that the application must pass for all stakeholders to accept and sign off the application. When you identify a functional requirement, you can also specify a corresponding acceptance test that must be run to ensure that the requirements are met.

User Stories in Extreme Programming

In extreme programming projects, developers perform even less functional requirement analysis at the beginning of the project, compared with other development models. They create user stories, instead of use cases or user scenarios. A user story is a very broad example of an interaction between the application and a user, and it is often stated in a single sentence as the following example illustrates:

- Users can upload photos and give new photos a title and a description.

User stories contain just the minimal details to enable developers to estimate the effort involved in developing it. Extreme programmers discuss each user story with stakeholders just before they write code to implement each user story.

Question: If a customer asks you to ensure 95% availability, is this a functional requirement or a technical requirement?

Planning the Database Design

When you have a good understanding of the functional requirements and technical requirements of the proposed web application, you can start designing the physical implementation of the application. One of the most important physical objects to plan for is one or more databases. Although not all web applications use databases for information storage, they are an underlying object for a majority of sites and you will use them in most of your projects.

- Logical Modeling
- Physical Database Structure
- Working with DBAs
- Database Design in Agile and Extreme Programming

Logical Modeling

You can begin your data design at a high level by creating UML Domain Model diagrams and Logical Data Model (LDM) diagrams.

A domain model diagram, also known as a conceptual data model, shows the high-level conceptual objects that your web application manages. For example, in an e-commerce web application, the domain model includes the concepts of customers, shopping carts, and products. The domain model does not include details of the properties each concept has, but shows the relationships between concepts. Use the domain model to record your initial conversations with stakeholders.

In essence, an LDM is a domain model diagram with extra details added. You can use LDM diagrams to fill in more details, such as properties and data types, for the concepts that you defined in the domain model. Note that the objects in the LDM do not correspond to tables in the database. For example, the shopping cart object may display data from both the customer database and product database tables.

Physical Database Structure

You should consider the following database objects in your project plan:

- *Tables*. These are the fundamental storage objects in a database. When you define a table, you need to specify the columns that each table has. For each column, you must define a data type such as **integer**, **string**, usually the **nvarchar** type in SQL Server, or **date and time**. You should also define the primary key for the table—the value of this column uniquely identifies each record and is essential for defining the relationships with records in other tables.
- *Views*. These are common presentations of data in tables and are based on queries. For example, a view can join two tables, such as a products table and a stock levels table.
- *Stored procedures*. These are common sequences of database operations that you define in the database. Some operations are complex and might involve a complex transformation of the data. You can define a stored procedure to implement such a complex routine.
- *Security*. You need to consider how the web application will authenticate with the database server and how you will authorize access to each database table.

In UML, a physical data model is a diagram that depicts tables, columns, data types and relationships between tables.

Working with Database Administrators

Sometimes, the developer team has full control over the database that underlies the web application. This happens, for example, when the organization is small or when the web application has a separate database server with no business-critical data. However, in larger organizations or in projects where the database stores critical business information, there may be a dedicated team of database administrators (DBAs). These DBAs are usually highly skilled in database design and administration, and their job is to ensure data integrity based on the organization's data storage policy.

If your project database is administered by the DBA team, it is essential to communicate well with them. You need to consult with DBAs for their requirements. They frequently impose a list of technical requirements that other stakeholders may not understand. As you build and deploy the web application, DBAs are responsible for creating databases on the right servers or clusters and assigning permissions. DBAs are critical contributors in delivering the web application.

Database Design in Agile Development and Extreme Programming

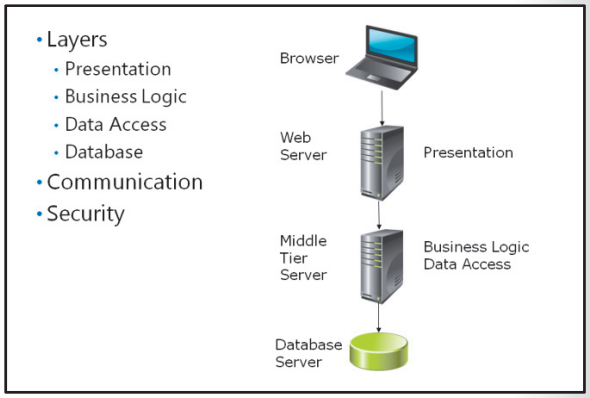
Agile development and extreme programming are characterized by a relatively small amount of initial planning and documentation, and acknowledge that requirements are likely to change throughout the development project. If you are working by using these development methodologies, you will only create domain models during the initial planning phase of your project. You do not develop LDMs or physical data models until you write code that implements the functional requirements. During the development phase, you will discuss requirements with users and DBAs and create LDMs and physical data models just before you write the code.

In agile development and extreme programming, the database design changes throughout the project until deployment. Therefore, developers should be able to alter the database whenever necessary without consulting DBAs or complying with complex data policies. For this reason, you should use a separate database hosted on a dedicated development server.

In some cases, the development project works with a database that already exists. For example, you may be developing a web application that presents and edits information from the company employee database on the intranet. In such cases, the database does not change as the code is developed, but functional and technical requirements may still change. You should copy the database to an isolated development database server to ensure that your developing code does not erroneously modify business-critical data.

Question: You want to implement a shopping cart in your web application. How many logical data models are required? How many database tables are required?

Planning for Distributed Applications



For small web application with less user traffic levels, you can choose to host all the components of your web application on a single server. However, as your web application grows, a distributed deployment, in which different servers host separate components of the application, is often used. Distributed web applications often use a layered architecture:

- *Presentation layer*. Components in this layer implement the user interface and presentation logic. If you are building an MVC web application, views and controllers make up your presentation layer.
- *Business logic layer*. Components in this layer implement high-level business objects such as products, or customers. If you are building an MVC web application, models make up your business logic layer.
- *Data access layer*. Components in this layer implement database access operations and abstract database objects, such as tables, from business objects. For example, a product business object may include data from both the Products and StockLevels database tables. If you are building an MVC web application, models often make up both business logic and data access layers. However, with careful design and coding practices, it is possible to refactor code to separate these layers.
- *Database layer*. This layer has the database itself.

If you implement such a layered architecture in your web application, you can host each layer on separate servers. Often, for example, the presentation layer is hosted on one or more IIS servers, the business logic and data access layers are hosted on dedicated middle-tier servers, and the database is hosted on a dedicated SQL Server. This approach has the following advantages:

- You can specify server hardware that closely matches each role. For example, a server that hosts business logic components requires good memory and processing resources.
- You can dedicate multiple servers to each role to ensure that a single server failure does not cause an interruption in service.
- Only the web servers must be on the perimeter network. Both middle-tier servers and database servers can be protected by two firewalls without direct access from the Internet.
- Alternatively, you can host middle-tier layers and databases on a cloud service, such as Windows Azure.

Communication Between Layers

When a single server hosts all the components of a web application, the presentation, business logic, and data access components run in a single process in the web server memory. Communication between components is not an issue. However, when you run different layers on different servers, you must consider two factors:

- How does each layer exchange information and messages?
- How does each server authenticate and secure communications with other servers?

The communication of information and security is performed in different ways between the various layers:

- *Between the browser and presentation layer web server.* In any web application, the web browser, where the presentation layer runs, communicates with the web server by using HTTP. If authentication is required, it is often performed by exchanging plain text credentials. You can also use Secure Sockets Layer (SSL) to encrypt this sensitive communication.
- *Between the web server and the middle-tier server.* The communication and security mechanisms used for communication between the web server and the middle-tier server depends on the technology that you use to build the business logic components:
  - Web services: If you implement business objects and data access classes as web services, the presentation layer components communicate with the web services by using HTTP. You can perform authentication by using the Kerberos protocol that is a part of Windows Integrated Authentication or by using plain text encrypted with SSL.
  - Windows Communication Foundation (WCF) services: If you implement business objects and data access classes as WCF services, you can choose between two hosting mechanisms. You can host the WCF services within IIS, in which case, HTTP is the transport mechanism and SSL is the security mechanism. You can also host the WCF services within a Windows Process Activation Service (WAS), in which case, you can use TCP, Microsoft Message Queuing (MSMQ), or named pipes as the transport mechanism.
- *Between middle-tier server and database.* The middle-tier server sends T-SQL queries to the database server, which authenticates against the database by using the required credentials that are often included in the connection string.

Note: Web services and WCF services are covered in detail in Course 20487A: Developing Windows Azure and Web Services.

Question: What are the advantages of writing middle-tier components as WCF services and not web services?

Planning State Management

In application development, the application state refers to the values and information that are maintained across multiple operations. Hypertext Transfer Protocol (HTTP) is fundamentally a stateless protocol, which indicates that it has no mechanism to retain state information across multiple page requests. However, there are many scenarios, such as the following, which require state to be preserved:

- Client-side locations to store state data:
  - Cookies
  - Query Strings
- Server-side locations to store state data:
  - TempData
  - Application State
  - Session State
  - Profile Properties
  - Database Tables

- *User preferences.* Some websites enable users to specify preferences. For example, a photo sharing web application might enable users to choose a preferred size for photos. If this preference information is lost between page requests, users have to continually reapply the preference.
- *User identity.* Some sites authenticate users to provide access to members-only content. If the user identity is lost between page requests, the user must re-enter the credentials for every page.
- *Shopping carts.* If the content of a shopping cart is lost between page requests, the customer cannot buy anything from your web application.

In almost all web applications, state storage is a fundamental requirement. ASP.NET provides several locations where you can store state information, and simple ways to access the state information. However, you must plan the use of these mechanisms carefully. If you use the wrong location, you may not be able to retrieve a value when you expect to. Furthermore, poor planning of state management frequently results in poor performance.

In general, you should be careful about maintaining large quantities of state data because it either consumes server memory, if it is stored on the server, or slows down the transfer of the webpage to the browser, if it is included in a webpage. If you need to store state values, you can choose between client-side state storage or server-side state storage.

Client-Side State Storage

When you store state information on the client, you ensure that server resources are not used. However, you should consider that all client-side state information must be sent between the web server and the web browser, and this process can slow down page load time. Use client-side state storage only for small amounts of data:

- *Cookies.* Cookies are small text files that you can pass to the browser to store information. A cookie can be stored:
  - In the client computer memory, in which case, it preserves information only for a single user session.
  - On the client computer hard disk drive, in which case, it preserves information across multiple sessions.

Most browsers can store cookies only up to 4,096 bytes and permit only 20 cookies per website. Therefore, cookies can be used only for small quantities of data. Also, some users may disable cookies for privacy purposes, so you should not rely on cookies for critical functions.

- *Query strings.* A query string is the part of the URL after the question mark and is often used to communicate form values and other data to the server. You can use the query string to preserve a small amount of data from one page request to another. All browsers support query strings, but some impose a limit of 2,083 characters on the URL length. You should not place any sensitive information in query strings because it is visible to the user, anyone observing the session, or anyone monitoring web traffic.

 **Note:** In ASP.NET Web Forms applications, View State, Control State, and Hidden Fields can be used to store state information in the rendered HTML that the server sends to the client. These mechanisms are not available in MVC web applications because they do not use Web Forms controls.

Server-Side State Storage

State information that is stored on the server consumes server resources, so you must be careful not to overuse server-side state storage or risk poor performance.

The following locations store state information in server memory:

- *TempData.* This is a state storage location that you can use in MVC applications to store values between one request and another. You can store values by adding them to the **TempData** collection. This information is preserved for a single request only and is designed to help maintain data across a webpage redirect. For example, you can use it to pass an error message to an error page.
- *Application State.* This is a state storage location that you can use to store vales for the lifetime of the application. The values stored in application state are shared among all users. You can store

values by adding them to the **Application** collection. If the web server or the web application is restarted, the values are destroyed. The **Application_Start()** procedure in the Global.asax file is an appropriate place to initialize application state values. Application state is not an appropriate place to store user-specific values, such as preferences, because if you store a preference in application state, all users share the same preference, instead of having their own unique value.

- *Session state.* The **Session** collection stores information for the lifetime of a single browser session and values stored here are specific to a single user session; they cannot be accessed by other users. By default, if the web server or the web application is restarted, the values are destroyed. However, you can configure ASP.NET to store session state in a database or state server. If you do this, session state can be preserved across restarts. Session state is available for both authenticated users and anonymous users. By default, session state uses cookies to identify users, but you can configure ASP.NET to store session state without using cookies.

If you choose to use these server memory locations, ensure that you estimate the total volume of state data that may be required for all the concurrent users that you expect to manage. Application state values are stored only once, but session state values are stored once for each concurrent user. Specify server hardware that can easily manage this load, or move state data into the following server hard disk drive-based locations.

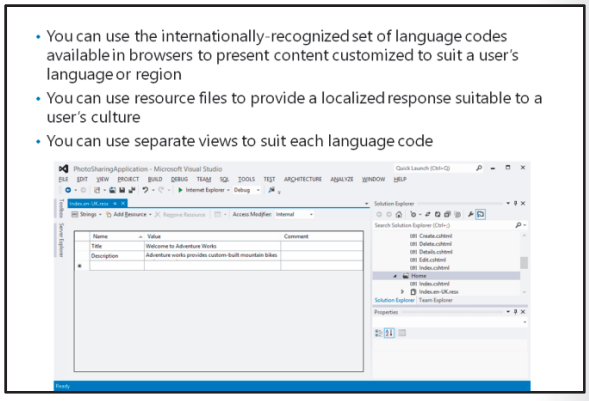
- *Profile properties.* If your site uses an ASP.NET profile provider, you can store user preferences in profiles. Profile properties are persisted to the membership database, so they will be kept even if the web application or web server restarts.
- *Database tables.* If your site uses an underlying database, like most sites do, you can store state information in its tables. This is a good place to store large volumes of state data that cannot be placed in server memory or on the client computer. For example, if you want to store a large volume of session-specific state information, you can store a simple ID value in the **Session** collection and use it to query and update a record in the database.

Remember that state data is only one form of information that an ASP.NET application places in server memory. For example, caches must share memory with state data.

Question: You show the visitors of your website a choice of countries. When they pick a state, you want to redirect them to a page that shows a map of that state. You will not use the name of the chosen country that the user selected after this. Which location should you use to store the name of the chosen country?

Planning Globalization and Localization

The Internet is an international network, and unless you are sure that the audience of your web application speaks a single language, you must ensure that everyone can read your content. You limit the site's potential audience if you render pages only in English. The process by which you make a web application available in multiple languages is called globalization or internationalization. The process by which you make a web application available in a specific language and culture is called localization.



Managing Browsers for Languages and Regions

There is an internationally-recognized set of language codes that specify a culture on the Internet. These codes are in two parts:

1. *The language*. For example, English is "en", and Russian is "ru".
2. *The region*. This specifies regional variations within a language and affects spellings and formats. For example, in United States English, "Globalize" is correct and dates are written in mm/dd/yy format, whereas in British English, "Globalise" is correct and dates are written in dd/mm/yy format.

The full language and region code for United States English is "en-US" and the full language and region code for British English is "en-UK".

The preferred language that users choose is available as the language code in the HTTP Header of the user's browser. This is the value that you respond to, so as to globalize your site. Alternatively, you can provide a control, such as a drop-down list, in which users can choose their preferred language. This is a good example of a user-preference that you can store in the session state.

Using Resource Files

When the user specifies a preferred language and region, you must respond by rendering pages for that culture. One method to provide a localized response is to use resource files to insert text in the appropriate language into the page at run time. A resource file is a simple dictionary of terms in a given language. For each term in the file, you need to specify a name, a value, and optionally, a comment. The file has an .resx extension. The file name also includes the language code that the resources apply to. For example, if you create a resource file for a view called, Index, which stored values in Chilean Spanish, you would name the file, Index.es-CL.resx.

Resource files can also have a neutral culture. This means that the file applies to any region in a given language. For example the Index.es.resx file applies Spanish terms, regardless of the regional code that has been chosen.

You should also create corresponding default resource files, without any language code in the file name, such as Index.resx. These files are used when a preferred language is not specified.

When you use resource files to localize a site, each view applies, regardless of the preferred language. You must insert extra Razor code in the view to take text values from a resource file. This can reduce the readability of view files because all the rendered text comes from resource files. However, supporting new languages is easier, because you only need to add a new resource file for each language that can be created by a professional translator.

Using Separate Views

Some developers prefer to use separate, parallel sets of views for each supported language code. If you use this approach, you must insert code into the controllers to detect the preferred language that the user has specified. Then, you can use this value to render the correct view.

When you use separate views to globalize and localize a site, views are more readable, because most of the text and labels remain in the view file. However, you must create view files, which requires you or your team members to be proficient in the target language.

Question: Which language do you consider would be appropriate to specify in the default resource file?

Lesson 2
Designing Models, Controllers, and Views

Models, controllers, and views are the fundamental building blocks of an MVC 4 web application. In a complex site, there may be hundreds of models, views, and controllers. You need to manage these objects and plan your application well, so that it is easy to manage the organization and internal structure during development. A thorough plan ensures that you detect any incorrect code and debug problems rapidly.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to design models.
- Describe how to design controllers.
- Describe how to design views.

Designing Models

A fundamental activity in the MVC design process is designing a model. Each model class within the model represents a kind of object that your application manages. You cannot plan for controllers and views until you understand the structure and design of the model.

- Model Classes and Properties
- Domain Model and Logical Data Model Diagrams
- Relationships and Aggregates
- Entity Framework
- Design in Agile and Extreme Programming

Identifying Model Classes and Properties

The use cases, usage scenarios, or user stories that you gathered during the analysis phase of the project should enable you to determine the model classes that you need to create. Each model class has a range of properties. For example, consider the following use case example shared earlier.

1. The user clicks the **Add Photo** link on the main site menu.
2. If the user is anonymous, the logon page is shown and the user provides credentials.
3. If the credentials are correct, the **CreatePhoto** view is displayed.
4. The user types a title.
5. The user specifies the photo file to upload.
6. The user optionally types a description for the photo.
7. The user clicks the **Upload** button.
8. The web application stores the new photo and displays the photo gallery to the user.

This example includes the following objects, each of which requires a model class:

- *User*. The User model class must include properties to store credentials, such as the user name and the password.
- *Photo*. The Photo model class must include the **Title** and **Description** properties.

Other use cases similarly help you add new model classes or new properties to the User and Photo model classes.

Domain Model and Logical Data Model Diagrams

You can use Domain Model diagrams and Logical Data Model (LDM) diagrams to analyze the information your website manages and suggest a physical data model or database design. You can return to these diagrams to plan model classes. Each object in your Domain Model or LDM diagram should be implemented as an MVC model class. The Domain Model diagram includes not only the names of those classes, but also the LDM diagram, if you have created one, with property names and data types.

Relationships and Aggregates

When you identify the model classes that you will implement in your website, you must also consider the relationships between them. For example, in the use case of the sample Photo Sharing application, each photo is associated with one, and only one, user. This is known as a one-to-one relationship. Each user, however, can create multiple photos. This is known as a one-to-many relationship.

Domain Model diagrams and LDM diagrams include such relationships as links between objects. Numbers at the ends of each link show whether the relationship is one-to-one, one-to-many, or many-to-many.

Aggregates place further limits on the behavior of model classes and clarify relationships. For example, in a photo sharing application, a photo is created by a single user. Other users can make multiple comments on each photo. If a user deletes a photo, all the comments on that photo should also be deleted from the database. However, the user who created the photo should not be deleted with the photo because he or she may add other photos or comments on the site. In this case, comments and photos should be placed in an aggregate, but users should be outside the aggregates. The photo is the "root entity" of the aggregate—this means that deleting a photo deletes all the associated comments, but deleting a comment does not delete the associated photo.

Entity Framework

Entity Framework is an Object Relational Mapping (ORM) framework for .NET Framework-based applications. An ORM framework links database tables and views to classes that a developer can program against, by creating instances or calling methods. Entity Framework has been a part of ADO.NET since .NET Framework 3.5.

When you use Entity Framework in your MVC web application, it links tables and views to the model classes that you have planned. You do not need to write SQL code to query or update database tables because Entity Framework does this for you. Entity Framework is well integrated with the Language Integrated Query (LINQ) language.

If you plan to use Entity Framework for data access, you should decide on how the database will be created during the planning phase:

- *Database-First*. Use the Entity Framework in the database-first mode when you have a pre-existing database to work with. This may happen because you already have data from an earlier system or because a DBA has designed the database for you. You can also choose this mode if you are familiar with creating databases in a database administration tool, such as Microsoft SQL Server Management Studio. When you use this mode, you have to specify the database connection string. Entity Framework connects to the database and examines the database schema. It creates a set of classes for you to use for data access.
- *Model-First*. Use Entity Framework in the model-first mode when you do not have a pre-existing database and prefer to design your model in Visual Studio. You can use the Entity Designer tool to name, configure, and link your model classes. This creates XML files that Entity Framework uses both to create model classes and to create the database with its tables and relationships.

- *Code-First*. Use Entity Framework in the code-first mode when you have no pre-existing database and prefer to design your models entirely in C# code. Your code must include **DBContext** and **DBSet** objects—these correspond to the database and its tables. When you run the application for the first time, Entity Framework creates the database for you.

Design in Agile and Extreme Programming

Agile and Extreme Programming projects are characterized by short design phases in which data models are not completed. Instead, a simple design, with little detail, is created and developers fill in details as they build code by continuously discussing requirements with users and other stakeholders.

In an MVC project, this means that you identity the model names and relationships during the design phase. You can record these on a Domain Model UML diagram. However, you can leave details such as property names and data types to be finalized in the development phase, along with the complete LDM diagrams.

Entity Framework lets you work in the Agile or Extreme Programming styles. For example, the framework can update the database when the model changes. Alternatively, the framework can update the model when the database changes. Entity Framework can perform these updates in any Entity Framework mode.

Designing Controllers

In an ASP.NET MVC web application, controllers are .NET Framework-based classes that inherit from the **System.Web.Mvc.Controller** base class. They implement input logic—this means that they receive input from the user in the form of HTTP requests and select both the correct model and view to use, to formulate a response.

Controller	Action
Photo	AddPhoto (GET)
	AddPhoto (POST)
	DisplayGallery (GET)
User	Logon (GET)
	Logon (POST)

- Identify Controllers and Actions
- Design in Agile and Extreme Programming

Identify Controllers and Actions

In an ASP.NET MVC web application, there is usually one controller for each model class. Conventionally, if the model class is called "Photo", the controller is called "PhotoController". If you follow this convention in your design, you can use the MVC default routing behavior to select the right controller for a request.

However, for each controller there can be many actions—each action is implemented as a method in the controller and usually returns a view. You often require separate actions for GET and POST HTTP request verbs. Similar to designing a model, you can identify the actions to write in each controller by examining the use cases you gathered during analysis. For example, consider the following use case shared earlier.

1. The user clicks the **Add Photo** link on the main site menu.
2. If the user is anonymous, the logon page is shown and the user provides credentials.
3. If the credentials are correct, the **CreatePhoto** view is displayed.
4. The user types a title.
5. The user specifies the photo file to upload.
6. The user optionally types a description for the photo.
7. The user clicks the **Upload** button.
8. The web application stores the new photo and displays the photo gallery to the user.

You have already identified Photo and User model classes from this use case. Adhering to the MVC convention, you should create a **PhotoController** and a **UserController**. The use case shows that the following actions are needed for each controller.

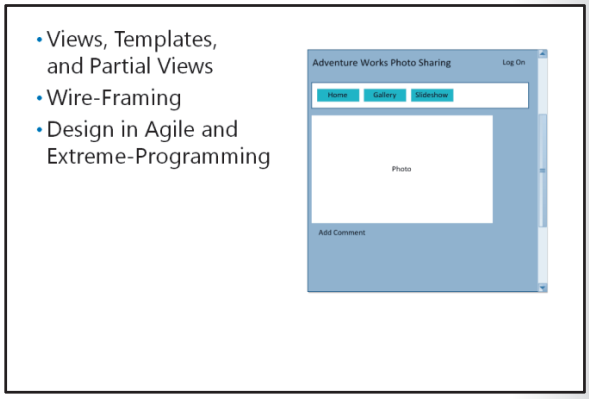
Controller	Action	Description
Photo	**AddPhoto (GET)**	The **AddPhoto** action for GET requests creates a new instance of the Photo model class, sets default values such as the created date, and passes it to the correct view.
	AddPhoto (POST)	The **AddPhoto** action for POST requests calls the Photo model class methods to save the photo values to the database and redirects the browser to the **DisplayGallery** action.
	DisplayGallery (GET)	The **DisplayGallery** action for GET requests displays all the photos stored in the database.
User	**Logon (GET)**	The **Logon** action for GET requests displays a view that an anonymous user can enter credentials into.
	Logon (POST)	The **Logon** action for POST requests checks user credentials against the membership database. If the credentials are correct, the **Logon** action authenticates and redirects the user to the originally requested page.

Design in Agile and Extreme Programming

Similar to the design of models, you will only make generalized plans for controllers during the design phase of an Agile Development or Extreme Programming project. Because you have identified the model classes necessary, you can plan the required controllers. However, you should specify only a few actions at this stage.

Designing Views

The user interface is a vital component of any system because it is the part that the users, budget holders, and other stakeholders see and interact with. Users are most interested in getting this part of the application right and frequently have the most to say about its design. As a developer, you have a chance to impress your users by designing and implementing a sophisticated user interface, which may result in more business.



In an ASP.NET MVC web application, the user interface is created by building views.

Views, Template Views, and Partial Views

There is a many-to-one relationship between MVC controllers and views. For example, a controller may use one view to display a single photo, another view to display several photos, and a third view to enable users to upload new photos. Each view corresponds to a webpage that the application can display to the user, although it can display different data. For example, the PhotoDetails view can display different photos, depending on the ID parameter that is passed to it.

As you plan views, you should also consider parts of the user interface that appear on all pages. For example, the company logo, main site menu, links to legal information, and logon controls may need to appear on every page. You can place these user interface components in a template view to create a consistent look and feel across pages.

 **Note:** Template views in ASP.NET MVC web applications perform the same role as master pages in ASP.NET Web Forms applications.

Some user interface components do not appear on all pages, but are re-used on several pages. For example, comments may be displayed in a single photo display, on the gallery, or on other pages. By creating a partial view, you can create a re-usable user interface element that can appear in many locations in this manner, without duplicating code.

Creating Wireframes

A common technique to discuss and plan the user interface for your application is to create wireframe diagrams. These are simplified layouts that show where the elements will appear on the final webpages. They are intended to communicate to users the essential functional parts of the user interface, but do not include graphics, logos, or colors. It is not necessary to create a wireframe model for every view in your application, but for only the most important ones.

You can begin a wireframe diagram by sketching it on a whiteboard, in conversation with a user. Many tools are available to create more formal versions—for example, Microsoft Visio has excellent wireframe drawing capabilities.

Design in Agile and Extreme Programming

You do not design many parts of the user interface during the initial phases of Agile Development or Extreme Programming projects. Instead, you design views and partial views in close consultation with users during the development phase. This applies even to the template view that displays common components of your user interface, although it is likely that the template view is one of the first user interface components that is designed and created. You will create the template view during the early iterations of the development phase of the project.

Lab: Designing ASP.NET MVC 4 Web Applications

Scenario

Your team has chosen ASP.NET MVC 4 as the most appropriate ASP.NET programming model to create the photo sharing application for the Adventure Works web application. You need to create a detailed project design for the application, and have been given a set of functional and technical requirements with other information. You have to plan:

- An MVC model that you can use to implement the desired functionality.
- One or more controllers and controller actions that respond to users actions.
- A set of views to implement the user interface.
- The locations for hosting and data storage.

Objectives

After completing this lab, you will be able to:

- Design an ASP.NET MVC 4 web application that meets a set of functional requirements.
- Record the design in an accurate, precise, and informative manner.

Estimated Time: 40 minutes

Virtual Machine: **20486B-SEA-DEV11**

User name: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Exercise 1: Planning Model Classes

Scenario

You need to recommend an MVC model that is required to implement a photo sharing application. You will propose model classes based on the results of an initial investigation into the requirements.

The main tasks for this exercise are as follows:

1. Examine the initial investigation.
2. Plan the photo model class.
3. Plan the comment model class.

▶ Task 1: Examine the initial investigation.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **InitialInvestigation** document by using the following information:
   - File location: **Allfiles (D):\Labfiles\Mod02**
3. Enable the **Navigation Pane** feature.
4. Read the contents of the Introduction section.

5. Read the contents of the General Description section.
6. Read the Use Cases section and then examine the **Use Case Summary** diagram.
7. Close the **InitialInvestigation** document.

▶ Task 2: Plan the photo model class.

1. Open the **DetailedPlanningDocument** document and locate the MVC Model section.
2. Based on your reading of the **InitialInvestigation** document, add and describe a model class for photos in **Table 1: MVC Model**.
3. Add properties to the model class you created in **Table 1: MVC Model**. The model class will have many properties
4. Add data types to the photo properties. Each property will have one and only one data type.
5. Merge the rows in the **Model Class** and **Description** columns and save the document.
6. Create a new UML Logical Data Model diagram in Visio 2010.
7. Add a new Class shape to model photos in the UML diagram.
8. Add attributes to the new Class shape for each of the properties you planned for the photos.
9. Save the created diagram by using the following information:
   - Folder path: **Allfiles (D):\Labfiles\Mod02**
   - File name: **PhotoSharingLDM**

▶ Task 3: Plan the comment model class.

1. Open the **DetailedPlanningDocument** document and locate the MVC Model section.
2. Based on your reading of the **InitialInvestigation** document, add and describe a model class for photos in **Table 1: MVC Model**.
3. Add properties to the model class you created in **Table 1: MVC Model**.
4. Add data types to the comment properties.
5. Merge the rows in the **Model Class** and the **Description** columns, and then save the document.
6. Add a new Class shape to model comments in the UML diagram.
7. Add attributes to the new Class shape for each of the properties you planned for comments.
8. In the UML diagram, connect the two class shapes.
9. Hide the end names for the connector.
10. Set multiplicity for the ends of the connector, and save the diagram.

Results: After completing this exercise, you will be able to create proposals for a model, and configure the properties and data types of the model classes.

Exercise 2: Planning Controllers

Scenario

You need to recommend a set of MVC controllers that are required to implement a photo sharing application. You will propose controllers based on the results of an initial investigation into the requirements.

The main tasks for this exercise are as follows:

1. Plan the photo controller.
2. Plan the comment controller.

▶ Task 1: Plan the photo controller.

1. Open the **DetailedPlanningDocument** document and locate the MVC Controllers section.
2. Based on your reading of the **InitialInvestigation** document, add a controller for photos in **Table 2: MVC Controllers**.
3. Add actions to the controller for photos in **Table 2: MVC Controllers**.
4. Add descriptions for each of the actions you have planned.
5. Merge rows in the **Controller** column and save the document.

▶ Task 2: Plan the comment controller.

1. Based on your reading of the **InitialInvestigation** document, add a controller for photos in **Table 2: MVC Controllers**.
2. Add actions to the controller for comments in **Table 2: MVC Controllers**.
3. Add descriptions for each of the actions you have planned.
4. Merge rows in the **Controller** column and save the document.

Results: After completing this exercise, you will be able to create proposals for controllers and configure their properties and data types.

Exercise 3: Planning Views

Scenario

You need to recommend a set of MVC views that are required to implement a photo sharing application. You will propose views based on the results of an initial investigation into the requirement.

The main tasks for this exercise are as follows:

1. Plan the single photo view.
2. Plan the gallery view.

▶ Task 1: Plan the single photo view.

1. Add a controller to the **Table 3: MVC Views** table.
2. Add the required views to the Controllers.
3. Add a description to the views.
4. Merge rows in the **Controller** column and save the document.
5. Create a new wireframe diagram in Visio 2010.
6. Add a new Application Form shape to the wireframe diagram.
7. Add a menu to the wireframe diagram.
8. Add a panel for the photo to the wireframe diagram.
9. Save the diagram by using the following information:

- File location: **Allfiles (D):\Labfiles\Mod02**
- File name: **SinglePhotoWireframe**

▶ Task 2: Plan the gallery view.

1. Create a new wireframe diagram in Visio 2010.
2. Add a new Application Form shape to the wireframe diagram.
3. Add a menu to the wireframe diagram.
4. Add multiple panels to the photo to the wireframe diagram.
5. Save the diagram by using the following information:
   - File location: **Allfiles (D):\Labfiles\Mod02**
   - File name: **PhotoGalleryWireframe**

Results: After completing this exercise, you will create proposals for views and their layouts.

Exercise 4: Architecting an MVC Web Application

Scenario

You need to recommend a web server and database server configuration that is required to implement a photo sharing application. You will propose details based on the results of an initial investigation into the requirements.

The main tasks for this exercise are as follows:

1. Hosting options.
2. Choose a data store.

▶ Task 1: Hosting options.

1. Based on your reading of the **InitialInvestigation** document, add a description of the web server arrangements that are suited to host the photo sharing application.

▶ Task 2: Choose a data store.

1. Based on your reading of the **InitialInvestigation** document, add a description of the database server arrangements that are suited to host the photo sharing application.

Results: After completing this exercise, you will be able to create proposals for hosting arrangements.

Question: What model classes should be created for the photo sharing application based on the initial investigation?

Question: What controllers should be created for the photo sharing application based on the initial investigation?

Question: What views should be created for the photo sharing application?

Module Review and Takeaways

In this module, you have seen how teams of developers, software architects, and business analysts collaborate to design an MVC web application that meets the needs of users. You can gather functional and technical requirements by talking to stakeholders and creating use cases, usage scenarios, and user stories. The model, view, controller, and other aspects of the design depend on these requirements. You have also seen how these design activities are completed in projects that use the agile methodology or extreme programming.

Best Practice: In Agile Development and Extreme Programming projects, developers discuss with users and stakeholders throughout development to ensure that their code will meet changing requirements. Even if you are not formally using these methodologies, it is good practice to regularly communicate with users.

Best Practice: When you design an ASP.NET MVC web application, start with the model, and then plan controllers, actions, and views. The controllers, actions, and views that you create each depend on the model.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
When you create a very detailed project plan, much of your work is wasted when requirements change late in the project.	

Review Question(s)

Question: You want to support both English and Spanish in your web application. You have both Spanish-speaking and English-speaking developers and want to ensure that views remain readable as easily as possible. Should you use multiple view files or multiple resource files to globalize your site?

Real-world Issues and Scenarios

You should bear in mind that when you select a project methodology, few projects follow a neat plan in real situations. Of the methodologies described in this module, agile development and extreme programming are the most flexible and respond when plans change in the middle of development. However, even with these methodologies, changing circumstances result in wasted development time and your project budget should include a contingency to cope with such changes.

Furthermore, when working with agile development and extreme programming projects, project managers must take care to avoid project creep or scope-creep. This occurs when people add new requirements when development takes place. Project creep results in projects that are over-budget and late.

Tools

Microsoft Office Visio: You can use Visio to create all types of UML software design diagrams, including Domain Model diagrams and LDMs. You can also use it to create wireframes.

Visual Studio 2012: You can create class diagrams such as LDMs in Visual Studio 2012.

Module 03

Developing ASP.NET MVC 4 Models

Contents:

Module Overview

Most web applications need to interact with various types of data or objects. An e-commerce application, for example, helps manage products, shopping carts, customers, and orders. A social networking application might help manage users, status updates, comments, photos, and videos. A blog is used to manage blog entries, comments, categories, and tags. When you write an MVC web application, you create an MVC model to *model* the data for your web application. Within this model, you create a model class for each type of object. The model class describes the properties of each type of object and can include business logic that matches business processes. Therefore, the model is a fundamental building-block in an MVC application and a good place to start when you write code. Models interact with the information store that underlies your site, which is usually a database. Therefore, you need to understand data access techniques and technologies to write models. In this module, you will see how to create the code for models and access data by using Entity Framework and LINQ.

Objectives

After completing this module, you will be able to:

- Add a model to an MVC application and write code in it to implement the business logic.
- Create a new SQL Azure database to store data for the web application.

Lesson 1
Creating MVC Models

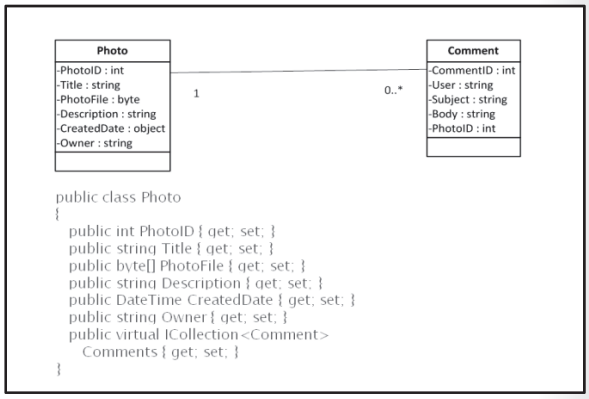
An MVC model is a collection of .NET Framework classes. When you create a model class, you define the properties and methods that suit the kind of object the model class describes. You can describe these properties in code so that MVC can determine how to render them in a webpage and how to validate user input. You need to know how to create and describe models, and how to modify the manner in which MVC creates model class instances when it runs your web application.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to create MVC models and develop business logic.
- Use the display and edit data annotations to assign attributes to views and controllers.
- Validate user input with data annotations.
- Describe model binders.
- Describe model extensibility.
- Add a model to an MVC 4 web application.

Developing Models



Every website presents information about various types of objects. In your web application, you need to define model classes for these objects. When you implement a functional requirement in your web application, you usually start by creating a model class. The model class will probably be revised when you create the controller and views for the model, and then later during the iterations that happen during the project. If you follow the Agile Development model or Extreme Programming, you begin with a simple understanding of the class—perhaps its name and a few properties. Then, you discuss with users and add details to the planned model class with the complete set of properties and its relationships to other classes. When developing the model, you can refer to use cases or user stories to ensure that these model details are correct.

When you feel that you fully understand the requirements for a model, you can write model classes to implement the requirements. The following lines of code illustrate how to create an example model class named **Photo**.

Example Model Class

```
public class Photo
{
   public int PhotoID { get; set; }
   public string Title { get; set; }
   public byte[] PhotoFile { get; set; }
   public string Description { get; set; }
   public DateTime CreatedDate { get; set; }
   public string Owner { get; set; }
```

```
    public virtual ICollection<Comment> Comments { get; set; }
}
```

Notice that the model class does not inherit from any other class. Also, notice that you have created public properties for each property in the model and included the data type, such as integer or string in the declaration. You can create read-only properties by omitting the **set;** keyword.

The **Photo** class includes a property called **Comments**. This is declared as a collection of **Comment** objects and implements one side of the relationship between photos and comments.

The following lines of code illustrate how you can implement the **Comment** model class.

Comment Model Class

```
public class Comment
    {
        public int CommentID { get; set; }
        public int PhotoID { get; set; }
        public string UserName { get; set; }
        public string Subject { get; set; }
        public string Body { get; set; }
        public virtual Photo Photo { get; set; }
    }
```

Notice that the **Comment** class includes a **PhotoID** property. This property stores the ID of the **Photo** that the user commented on, and it ties the comment to a single photo. Also, notice that the **Comment** class includes a **Photo** property, which returns the Photo object that the comment relates to. These properties implement the other side of the relationship between photos and comments. The instances of model classes are usually created in a controller action and passed to a view to display.

The following code example shows how a controller action can create a new photo object from the photo model class and pass it to the Display view.

Instantiating a Photo in a Controller Action

```
Photo newPhoto = new Photo();
newPhoto.Title = "This is an Example Photo";
newPhoto.Owner = User.Identity.Name;
newPhoto.CreatedDate = DateTime.Today;
return View("DisplayView", newPhoto);
```

In the Display view, you can render the Title property of a photo by using the **Model** object in the Razor code that you write, as the following code shows.

Rendering a Property from a Model Class

```
<div id="photo-title">
   @Model.Title
</div>
```

Using Display and Edit Data Annotations on Properties

The model classes usually specify three attributes for each property:

- The name of the property, for example, Title
- The data type of the property, for example, String
- The access levels of the property, for example, the **get** and **set** keywords to indicate read and write access

```
public class Photo
{
  public int PhotoID { get; set; }
  public string Title { get; set; }
  [DisplayName("Picture")]
  public byte[] PhotoFile { get; set; }
  [DataType(DataType.MultilineText)]
  public string Description { get; set; }
  [DataType(DataType.DateTime)]
  [DisplayName("Created Date")]
  [DisplayFormat(DataFormatString = "{0:MM/dd/yy}",
    ApplyFormatInEditMode = true)]
  public DateTime CreatedDate { get; set; }
  public string UserName { get; set; }
  public virtual ICollection<Comment>
    Comments { get; set; }
}
```

Additionally by using attributes, you can supply additional metadata to describe properties to ASP.NET MVC. The MVC runtime uses this metadata to determine how to render each property in views for displaying and editing. These attributes are called display and edit annotations.

For example, property names in C# cannot contain spaces. On a rendered webpage, you may often want to include spaces in a property label. For example, you might want to render a property called **CreatedDate** with the label Created Date. To provide MVC with this information, you can use the **DisplayName** annotation.

When you use MVC, you can indicate how you want a property to be named on a view by using the **DisplayName** annotation, as the following lines of code illustrate.

Setting the DisplayName Annotation

```
 [DisplayName("Created Date")]
public CreatedDate { get; set; }
```

If you have a **DateTime** property, you can use display and edit data annotations to inform MVC what format you want the property displayed in.

In the following lines of code, the **CreatedDate** property is a **DateTime** and the **DisplayFormat** data annotation is used to indicate to MVC that only the day, month, and year values should be displayed.

Setting the DataType and DisplayFormat

```
 [DisplayName("Created Date")]
[DataType(DataType.DateTime)]
[DisplayFormat(DataFormatString = "{0:MM/dd/yy}", ApplyFormatInEditMode = true)]
public DateTime CreatedDate { get; set; }
```

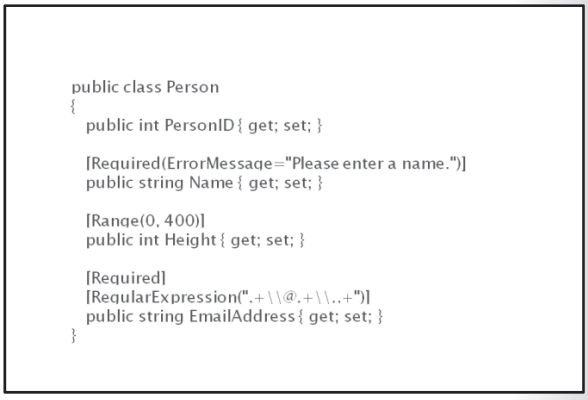
All the data annotations that are provided with ASP.NET MVC 4 are included in the **System.ComponentModel.DataAnnotations** namespace.

Additional Reading: To read more about all the data annotations provided with MVC 4, see http://go.microsoft.com/fwlink/?LinkID=288949&clcid=0x409

Question: In the code on the slide, how can you recognize the display and edit annotations and distinguish them from other code?

Validating User Input with Data Annotations

You can use data annotations in MVC models to set validation criteria for user input. Input validation is the process by which MVC checks data provided by a user or a web request to ensure that it is in the right format. The following example shows a webpage form that collects some information from the user:



- *Name*. This is required input. The user must enter some data in this field.
- *Height*. This must be an integer between 0 and 400.
- *Email Address*. This is required input. The value entered must be a valid email address.

In the following example, when the user submits the form, you want MVC to create a new instance of the **Person** model and use it to store the data in the database. However, you want to ensure that the data is valid before it is stored.

To ensure that the data is valid, you can define the **Person** model class by using the following lines of code.

Using Validation Data Annotations

```
public class Person
{
   public int PersonID { get; set; }
   [Required]
   public string Name { get; set; }
   [Range(0, 400)]
   public int Height { get; set; }
   [Required]
   [RegularExpression(".+\\@.+\\..+")]
   public string EmailAddress { get; set; }
}
```

The **Required**, **Range**, **StringLength**, and **RegularExpression** annotations implement input validation in MVC. If users do not enter data that satisfies the criteria specified for each property, the view displays a standard error message that prompts the user to enter the correct data. In the earlier example, you can see that the user must enter a **Name** and a **Height** between 0 and 400. For the **EmailAddress** property**,** the user must enter a value that matches the regular expression. The regular expression in the example is a simple expression that requires an @ symbol and a dot.

To specify the error message that the user sees when data is not valid, you can use the **ErrorMessage** property on the validation data annotations, as the following code illustrates.

Setting a Validation Error Message

```
 [Required(ErrorMessage="Please enter a name.")]
public string Name { get; set; }
```

Note: You will see how to ensure that the validation error message is displayed in a view.

Additional Reading: For more examples of validation data annotations see http://go.microsoft.com/fwlink/?LinkID=288950&clcid=0x409

Additional Reading: For more information about the regular expressions that you can use to check user input, see http://go.microsoft.com/fwlink/?LinkID=288951&clcid=0x409

Question: You want to ensure that users enter a password that is longer than six characters. How should you do this by using a validation data annotation?

What Are Model Binders?

A model binder is a component of an ASP.NET MVC application that creates an instance of a model class, based on the data sent in the request from the web browser. ASP.NET MVC includes a default model binder that meets the needs of most web applications. However, you must know how the default model binder works with other components to use it properly. In addition, you may choose to create a custom model binder for advanced situations.

- The Default Controller Action Invoker uses model binders to determine how parameters are passed to actions
- The Default Model Binder passes parameters by using the following logic:
  - The binder examines the definition of the action that it must pass parameters to
  - The binder searches for values in the request that can be passed as parameters

What Does a Model Binder Do?

A model binder ensures that the right data is sent to the parameters in a controller action method. This enables MVC to create instances of model classes that satisfy the user's request. The default model binder, for example, examines both the definition of the controller action parameters and the request parameters to determine which request values to pass to which action parameter.

This model binding process can save developers a lot of time and avoid many unexpected run-time errors that arise from incorrect parameters. MVC includes a default model binder with sophisticated logic that passes parameters correctly in almost all cases without complex custom code.

The Controller Action Invoker and the Default Model Binder

To understand the default model binding process, consider the following request from a web browser:

http://www.adventureworks.com/product/display/45

This request identifies three aspects:

- The model class that interests the user. The user has requested a product.
- The operation to perform on the model class. The user has requested that the product be displayed.
- The specific instance of the model class. The user has requested that the product with ID 45 be displayed.

The request is received by an object called the controller action invoker. The controller action invoker of the MVC runtime calls a controller action and passes the correct parameters to it. In the example, the action invoker calls the **Display** action in the **Product** controller and passes the ID "45" as a parameter to the action, so that the right product can be displayed.

The **ControllerActionInvoker** class is the default action invoker. This action invoker uses model binders to determine how parameters are passed to actions.

How the Default Model Binder Passes Parameters

In a default MVC application, there is only one model binder for the **ControllerActionInvoker** to use. This binder is an instance of the **DefaultModelBinder** class. The default model binder passes parameters by using the following logic:

1. The binder examines the definition of the action that it must pass parameters to. In the example, the binder determines that the action requires an integer parameter called **PhotoID**.
2. The binder searches for values in the request that can be passed as parameters. In the example, the binder searches for integers because the action requires an integer. The binder searches for values in the following locations, in order:
    a. *Form Values*. If the user fills out a form and clicks a submit button, you can find parameters in the **Request.Form** collection.
    b. *Route Values*. Depending on the routes that you have defined in your web application, the model binder may be able to identify parameters in the URL. In the example URL, "45" is identified as a parameter by the default MVC route.
    c. *Query Strings*. If the user request includes named parameters after a question mark, you can find these parameters in the **Request.QueryString** collection.
    d. *Files*. If the user request includes uploaded files, these can be used as parameters.

Notice that if there are form values and route values in the request, form values take precedence. Query string values are only used if there are no form values and no route values available as parameters.

Model Extensibility

The MVC architecture has been designed to provide extensibility so that developers can adapt the architecture to unusual or unique requirements. For example, the default action invoker, **ControllerActionInvoker**, can be replaced by your own action invoker if you want to implement your own invoking logic.

Two ways in which you can extend the MVC handling of MVC models are to create custom data annotations and custom model binders.

Custom validation data annotations can be used to indicate to MVC how to validate the data a user enters in a form or passes in query strings

- There are four built-in validation attributes:
    - Required
    - Range
    - StringLength
    - RegularExpression

A custom model binder ensures that it identifies parameters in a request and passes all of them to the right parameters on the action

Custom Validation Data Annotations

You can use data annotations to indicate to MVC how to validate the data that a user enters in a form or passes in query strings. The four built-in validation attributes in MVC 4, **Required**, **Range**, **StringLength**, and **RegularExpression**, are very flexible. However, in some situations, such as the following examples, you may want to run some custom validation code:

- *Running a Data Store Check*. You want to check the data entered against data that has already been stored in the database or in another database store.
- *Comparing Values*. You want to compare two entered values with each other.
- *Mathematical Checks*. You want to calculate a value from the entered data and check that value is valid.

In such situations you can create a custom validation data annotation. To do this, you create a class that inherits from the **System.ComponentModel.DataAnnotations.ValidationAttribute** class.

The following lines of code illustrate how to create a custom validation data annotation.

Creating Custom Validation Data Annotation

```
 [AttributeUsage(AttributeTargets.Field)]
public class LargerThanValidationAttribute : ValidationAttribute
{
   public int MinimumValue { get; set; }
   //Constructor
   public LargerThanValidationAttribute (int minimum)
   {
      MinimumValue = minimum;
   }
   //You must override the IsValid method to run your test
   public override Boolean IsValid (Object value)
   {
      var valueToCompare = (int)value;
      if (valueToCompare > MinimumValue)
      {
         return true;
      }
      else
      {
         return false;
      }
   }
}
```

After you have created a custom validation data attribute, you can use it to annotate a property in your model, as the following lines of code illustrate.

Using a Custom Validation Data Annotation

```
 [LargerThanValidationAttribute(18)]
public VoterAge { get; set; }
```

Custom Model Binders

The default controller action invoker receives requests and calls the right action on the right controller to fulfill the request. The default action invoker uses model binders to identify parameters in the request and pass all of them to the right parameters in the action. This mechanism ensures that query strings, route values, form values, and uploaded files are available for the action to work with.

The default model binder is sophisticated and flexible, but sometimes, you may want to customize its behavior to pass parameters in an unusual way. You can do this by creating a custom model binder and registering it with the MVC runtime.

The following lines of code show how to create a simple custom model binder to pass parameters from the form collection by implementing the **IModelBinder** interface.

A Simple Custom Model Binder

```
public class CarModelBinder : IModelBinder
{
   public object BindModel (ControllerContext controllerContext, ModelBindingContext
bindingContext)
   {
      //Get the color for the car from the request form
      string color = controllerContext.HttpContext.Request.Form["color"];
      //Get the brand for the car from the request form
      string brand= controllerContext.HttpContext.Request.Form["brand"];
      //Create a new instance of the car model
      Car newCar = new Car();
      newCar.color = color;
      newCar.brand = brand;
```

```
        //return the car
        return newCar;
    }
}
```

The code example assumes that you have a model class in your MVC application called, Car. It also assumes that any request for a Car object includes values for **color** and **brand** in the form collection. This situation can easily be handled by the default model binder. However, this example demonstrates how model binders can locate values from the context of the request and pass those values to the right properties in the model. You can add custom code to implement extra functionality.

 **Additional Reading:** You can see more examples of custom model binders at the following locations:

- http://go.microsoft.com/fwlink/?LinkID=288952&clcid=0x409
- http://go.microsoft.com/fwlink/?LinkID=288953&clcid=0x409

Question: You want to ensure that when a user types a value into the Car Model Number box when adding a new car to the website, the text entered is not already used by another car in the database. Would you use a custom validation data annotation or a custom model binder for this?

Demonstration: How to Add a Model

In this demonstration, you will see how to create a model in an ASP.NET MVC 4 web application and add model classes to the web application. You will also see how to add data annotations and create a custom validation data annotation.

Demonstration Steps

1. On the **File** menu of the **Start Page - Microsoft Visual Studio** window, point to **New,** and then click **Project.**
2. In the navigation pane of the **New Project** dialog box, expand **Installed**, expand **Templates**, and then expand **Visual C#**.
3. Under Visual C#, click **Web**, and then, in the result pane, click **ASP.NET MVC 4 Web Application**.
4. In the **Name** box of the **New Project** dialog box, type **OperasWebSites**.
5. In the **New Project** dialog box, click **Browse**.
6. In the **Location** text box, navigate to **Allfiles (D):\Democode\Mod03**, and then click **Select Folder**.
7. In the **New Project** dialog box, click **OK**.
8. In the **Select a Template** list of the **New ASP.NET MVC 4 Project** dialog box, click **Empty**, and then click **OK**.
9. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, right-click **Models**, point to **Add**, and then click **Class**.
10. In the **Name** box of the **Add New Item - OperasWebSites** dialog box, type **Opera.cs**, and then click **Add**.
11. In the Opera class of the Opera.cs code window, type the following code.

```
public int OperaID { get; set; }
public string Title { get; set; }
```

```
public int Year { get; set; }
public string Composer { get; set; }
```

12. Place the mouse cursor at the end of the **OperaID** property code, press Enter, and then type the following code.

```
[Required]
[StringLength(200)]
```

13. In the Required data annotation, right-click **Required**, point to **Resolve**, and then click **using System.ComponentModel.DataAnnotations**.

14. Place the mouse cursor at the end of the **Year** property, press Enter, and then type the following code.

```
[Required]
```

15. Place the mouse cursor at the end of the Opera class, press Enter, and then type the following code.

```
public class CheckValidYear : ValidationAttribute
{
}
```

16. In the CheckValidYear class, type the following code.

```
public override bool IsValid(object value)
{
   int year = (int)value;
   if (year < 1598)
   {
      return false;
   }
   else
   {
      return true;
   }
}
```

17. In the CheckValidYear class, type the following code.

```
public CheckValidYear()
{
   ErrorMessage = "The earliest opera is Daphne, 1598, by Corsi, Peri, and
Rinuccini";
}
```

18. In the Opera class, place the mouse cursor at the end of the **Title** property code, press Enter, and then type the following code.

```
[CheckValidYear]
```

19. On the **Build** menu of the **OperasWebSites - Microsoft Visual Studio** window, click **Build Solution**, and then note that the application is being built.
20. In the **OperasWebSites - Microsoft Visual Studio** window, click the **Close** button.

Lesson 2
Working with Data

All web applications present information and almost all web applications require a data store for that information. By rendering webpages by using data from a data store, you can create a web application that changes continually in response to user input, administrative actions, and publishing events. The data store is usually a database, but other data stores are occasionally used. In MVC applications, you can create a model that implements data access logic and business logic. Alternatively, you can separate business logic, in model classes, from data access logic, in a repository. A repository is a class that a controller can call to read and write data from and to a data store. The .NET Framework includes the Entity Framework and LINQ technologies, which make data access code very quick to write and simple to understand. In addition, you will see how to build a database-driven website in MVC.

Lesson Objectives

After completing this lesson, you will be able to:

- Connect an application to a database to access and store data.
- Describe the features of the Entity Framework.
- Use LINQ to write queries for selecting, filtering, and grouping data.
- Create separate model classes and corresponding repository classes by using Entity Framework code.
- Explain how to access data in models and repositories.

Connecting to a Database

Most websites use a database to store dynamic data. By including this data in rendered HTML pages, you can create a dynamic web application with content that changes frequently. For example, you can provide administrative webpages that enable company employees to update the product catalog and publish news items. Products and items are stored in the database. As soon as they are stored, users can view and read them. The employees do not need to edit HTML or republish the website to make their changes visible.

ADO.NET supports a wide range of databases by using different data providers

Cloud Databases can be used for web applications that are hosted in the cloud

To connect an MVC web application to a database:
- Add a reference to the **System.Data** namespace
- Add a connection string to the **web.config** file

Some websites may store data in other locations, such as a directory service, but databases are the most widely used data store.

ADO.NET and Databases

When you create .NET Framework applications, including MVC web applications, you can use the ADO.NET technology to access databases. ADO.NET classes are contained in the **System.Data** namespace. ADO.NET supports a wide range of databases by using different data providers. For example:

- *Microsoft SQL Server*. This is an industry-leading database server from Microsoft. ADO.NET includes the **SqlClient** provider for all SQL Server databases.

- *Microsoft SQL Server Express*. This is a free version of SQL Server that includes a wide range of database functionality and is very flexible. Some advanced capabilities, such as database clustering, are not possible with SQL Express. The **SqlClient** provider is used for SQL Express.
- *Microsoft SQL Server Compact*. This version of SQL is also free and uses .sdf files to store data on the hard disk. ADO.NET includes the **SqlServerCe** provider for SQL Compact databases.
- *Oracle Databases*. This is a widely-used database server. ADO.NET includes the **OracleClient** provider for all Oracle databases.
- *OLE DB*. This is a standard that many different databases adhere to. ADO.NET includes the **OleDb** provider for all OLE DB databases.
- *ODBC*. This is another older standard that many different databases adhere to. ADO.NET includes the **Odbc** provider for all ODBC databases. In general, you should use an OLE DB provider, if it is available, instead of an ODBC provider.

You can also use third-party ADO.NET providers to access other databases.

Cloud Databases

The database of a web application is usually located on the same server as the web application itself or on a dedicated database server at the same physical site. However, if you have a fast, reliable Internet connection, you can consider using a cloud database. Furthermore, if you have chosen to host a web application in the cloud, a cloud database is a logical storage solution. The Microsoft cloud database service is called SQL Database and is a part of Windows Azure.

Windows Azure SQL Database has the following advantages:

- Databases run in Microsoft data centers with the best connectivity and reliability.
- Microsoft guarantees up to 99% uptime.
- You do not need to build and maintain your own database servers or employ database administrators.
- You can scale up the databases very easily.
- You pay only for the data that you use and distribute.

You can use Windows Azure SQL Database with ADO.NET by using the **SqlClient** provider.

Connecting an MVC Web Application to a Database

To use ADO.NET and connect to a database, you need to add two items to your application:

- Add a reference to the **System.Data** namespace.
- Add a connection string to the **Web.config** file. This string specifies the provider, the location of the database, the security properties, and other properties depending on the provider.

The following markup shows how to add a connection string to **Web.config** to connect to a SQL Express database called, PhotoSharingDB, by using the credentials that the web application runs under.

Connecting to SQL Express

```
<connectionStrings>
   <add name="PhotoSharingDB"
      connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=PhotoSharingDB;" +
         "Integrated Security=SSPI"
      providerName="System.Data.SqlClient"/>
</connectionStrings>
```

The following connection string connects to a Windows Azure SQL database.

Connecting to Windows Azure SQL Database

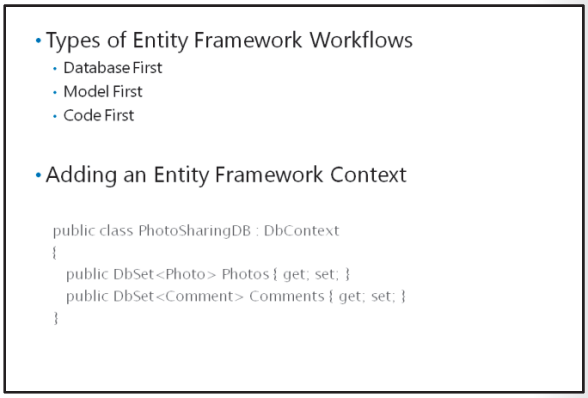
```
<connectionStrings>
   <add name="PhotoSharingDB"

connectionString="Server=tcp:example.database.windows.net,1433;Database=PhotoSharingDB;"
+
"User ID=Admin@example;Password=Pa$$w0rd;Trusted_Connection=False;" +
         "Encrypt=True;Connection Timeout=30;PersistSecurityInfo=true"
      providerName="System.Data.SqlClient"/>
</connectionStrings>
```

The **<connectionStrings>** tag must appear within the **<configuration>** tag, after the **<configSections>** tag.

Additional Reading: The MVC web application templates in Microsoft Visual Studio 2012 include ADO.NET references so you need not add them. However, you must add connection strings. Some of the templates, such as the Internet site template, include SQL Server Express membership databases with connection strings.

The Entity Framework

Developers write code that works with classes and objects. By contrast, databases store data in tables with columns and rows. Database administrators create and analyze databases by running Transact-SQL queries. You can choose to build and run Transact-SQL queries in your ADO.NET code. However, ADO.NET includes the Entity Framework—this technology enables you to read and write data to and from a database by using classes and objects.



Entity Framework is an Object Relational Mapping (ORM) framework. An ORM framework maps the tables and columns found in a database to objects and their properties that you can call from .NET code.

Entity Framework Workflows

The way you use Entity Framework in your application depends on the manner in which you want to build your database. The three Entity Framework workflows available are database-first, model-first, and code-first:

- *Database First*. You can use the database-first workflow when you have a pre-existing database or if you prefer to create a new database by defining table, columns, views, and other database schema objects. In this workflow, Entity Framework examines the database and creates an XML file with an .edmx extension called the model file. The model file describes classes that you will be able to work with, in code. You can adjust the model by using a designer in Visual Studio and then writing code against the Entity Framework classes.
- *Model First*. You can use the model-first workflow when you do not yet have a database and you prefer to design your model by using an ORM modeling tool. In this workflow, you create the .edmx file in the Visual Studio designer and then write code against the model classes generated by the designer. When you run the application, Entity Framework creates the database tables and

columns to support the model. In this workflow, Visual Studio can also create a connection string for you and insert it into Web.config, based on the database that you specify.

- *Code First*. You can use the code-first workflow if you prefer to create a model by writing .NET Framework classes. In this workflow, there is no model file. Instead, you create model classes in C# or Visual Basic. When you run the application, Entity Framework creates the database.

Adding an Entity Framework Context

When you use Entity Framework in the code-first workflow, you must ensure that the framework creates the right database and tables to store your model classes. To do this, create a class that inherits the Entity Framework **DbContext** class. You will use this class in controllers when you want to manipulate data in the database. Within this class, add a **DbSet<>** property for each database table you want Entity Framework to create in the new database.

The following code shows how to add an Entity Framework context to your model.

An Entity Framework Context Class

```
public class PhotoSharingDB : DbContext
{
   public DbSet<Photo> Photos { get; set; }
   public DbSet<Comment> Comments { get; set; }
}
```

In the earlier example, Entity Framework looks for a connection string with the name, **PhotoSharingDB**, to match the name of the **DbContext** class. Entity Framework creates the database at the location that the connection string provides, and creates two tables in the new database:

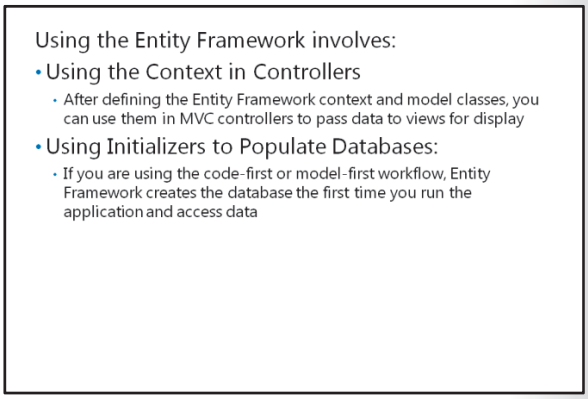
- **Photos:** This table will have columns that match the properties of the **Photo** model class.
- **Comments:** This table will have columns that match the properties of the **Comment** model class.

Question: You have a Microsoft Visio diagram, which a business analyst created that shows all the model classes for your web application and their relationships. You want to re-create this diagram in Visual Studio. Which Entity Framework workflow should you use?

Using an Entity Framework Context

Now that you have defined the Entity Framework context and model classes, you can use them in MVC controllers to pass data to views for display.

The following code shows how to use the Entity Framework context in a controller to pass a single photo, or a collection of photos, to views.



Using an Entity Framework Context in a Controller

```
public class PhotoController : Controller
{
   //Creating a reference to the Entity
Framework context class
   private PhotoSharingDB db = new PhotoSharingDB();
   //This action gets all the photos in the database and passes them to the Index view
   public ActionResult Index()
   {
      return View("Index", db.Photos.ToList());
   }
```

```
   //This action gets a photo with a particular ID and passes it to the Details view
   public ActionResult Details(int id = 0)
   {
      Photo photo = db.Photos.Find(id);
      if (photo == null)
      {
         return HttpNotFound();
      }
         return View("Details", photo);
      }
   }
}
```

Using Initializers to Populate Databases

If you are using the code-first or model-first workflow, Entity Framework creates the database the first time you run the application and access data. The database remains empty if you have created the database schema but not populated it with data rows.

You can use an initializer class to populate the database with sample data. This technique ensures that there is sample data to work with, during development.

The following lines of code show how to create an initializer class that adds two **Photo** objects to the Photos table in the database.

An Example Entity Framework Initializer

```
public class PhotoSharingInitializer : DropCreateDatabaseAlways <PhotoSharingDB>
{
   //Override the Seed method to populate the database
   protected override void Seed(PhotoSharingDB context)
   {
      //Create a list of Photo objects
      var photos = new List<Photo>
      {
         new Photo {
            Title = "My First Photo",
            Description = "This is part of the sample data",
            UserName = "Fred"
         },
         new Photo {
            Title = "My Second Photo",
            Description = "This is part of the sample data",
            UserName = "Sue"
         }
      };
      //Add the list of photos to the database and save changes
      photos.ForEach(s => context.Photos.Add(s));
      context.SaveChanges();
   }
}
```

After you have created an initializer, you need to ensure that it runs by adding a line of code to the **Global.asax** file in the **Application_Start** method, as the following example illustrates.

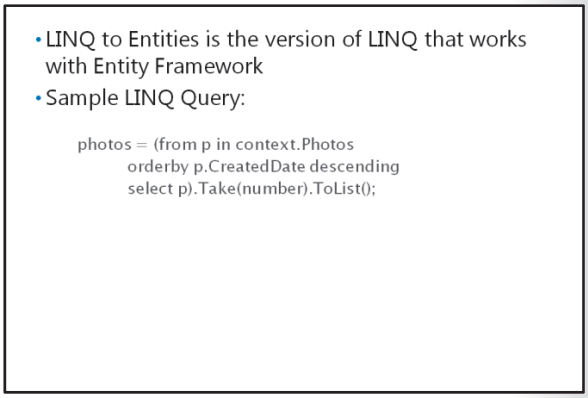
Running the Initializer in Global.asax

```
protected void Application_Start()
{
   //Seed the database with sample data for development. This code should be removed for
production.
   Database.SetInitializer<PhotoSharingDB>(new PhotoSharingInitializer());
}
```

Question: You have created an Entity Framework context class in your model, added an initialize, and called **Database.SetInitializer()** from Global.asax. When you run the application, no database is created and no model objects displayed on the webpages. What have you forgotten to do?

Using LINQ to Entities

Language Integrated Query (LINQ) is a set of extensions to Visual C# and Visual Basic that enable you to write complex query expressions. You can use these expressions to extract data from databases, enumerable objects, XML documents, and other data sources. The expressions are similar to Transact-SQL queries, but use C# or VB keywords so that you may get IntelliSense support and error checking in Visual Studio.



What Is LINQ to Entities?

LINQ to Entities is the version of LINQ that works with Entity Framework. LINQ to Entities enables you to write complex and sophisticated queries to locate specific data, join data from multiple objects, update data, and take other actions on objects from an Entity Framework context. If you are using Entity Framework, you can write LINQ queries wherever you require a specific instance of a model class, a set of objects, or for more complex application needs. You can write LINQ queries in query syntax, which resembles SQL syntax, or method syntax, in which operations such as "select" are called as methods on objects.

Example LINQ Queries

In the following lines of code, you can see how to obtain a list of the most recent photos in the database. Both query syntax and method syntax examples are included.

Using LINQ to Get Recent Photos

```
//This list will store the photos that are returned
List<Photo> photos;
//This is the Entity Framework context
PhotoSharingDB context = new PhotoSharingDB();
if (number == 0)
{
   //If a number of photos is not specified, we'll get all the photos in the database
   //This example is in method syntax.
   photos = context.Photos.ToList();
}
else
{
   //The number specifies how many of the most recent photos the user requests
   //Use a LINQ query with both query and method syntax to get these from the database
   photos = (from p in context.Photos
                          orderby p.CreatedDate descending
                          select p).Take(number).ToList();
}
```

Additional Reading: To read many more example LINQ queries, see http://go.microsoft.com/fwlink/?LinkID=288954&clcid=0x409

Demonstration: How to Use Entity Framework Code

In this demonstration, you will see how to add a connection string, an Entity Framework context, and an initializer to the web application.

Demonstration Steps

1. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, click **web.config**.
2. In the web.config code window, place the mouse cursor at the end of the **</appsettings>** tag, press Enter, and then type the following code.

```
<connectionStrings>
  <add name="OperasDB"
   connectionString=
   "Data Source=(LocalDB)\v11.0;" +
"AttachDbFilename=" +
   "|DataDirectory|\Operas.mdf;" +
   "Integrated Security=True"
   providerName=
    "System.Data.SqlClient" />
</connectionStrings>
```

3. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, right-click **References**, and then click **Manage NuGet Packages**.
4. In the **OperasWebSite - Manage NuGet Packages** window, click **Online**, click **EntityFramework**, and then click **Install**.
5. On the **License Acceptance** page, click **I Accept**.
6. In the **OperasWebSite - Manage NuGet Packages** window, click **Close**.
7. In the **Microsoft Visual Studio** dialog box, click **Yes to All**.
8. In the Solution Explorer pane, right-click **Models**, point to **Add**, and then click **Class**.
9. In the **Name** box of the **Add New Item - OperasWebSite** dialog box, type **OperasDB**, and then click **Add**.
10. In the OperasDB.cs code window, locate the following code.

```
using System.Web;
```

11. Place the mouse cursor at the end of the located code, press Enter, and then type the following code.

```
using System.Data.Entity;
```

12. In the OperasDB.cs code window, locate the following code.

```
public class OperaDB
```

13. Append the following code to the existing line of code.

```
: DbContext
```

14. In the **OperaDB** class, type the following code.

```
public DbSet<Opera> Operas
   { get; set; }
```

15. In the Solution Explorer pane, right-click **Models**, point to **Add**, and then click **Class**.

16. In the **Name** box of the **Add New Item - OperasWebSite** dialog box, type **OperasInitializer**, and then click **Add**.
17. In the OperasInitializer.cs code window, place the mouse cursor at the end of the System.web namespace code, press Enter, and then type the following code.

```
using System.Data.Entity;
```

18. In the OperasInitializer.cs code window, locate the following code.

```
public class OperasInitializer
```

19. Append the following code to the existing line of code.

```
: DropCreateDatabaseAlways
   <OperasDB>
```

20. In the **OperasInitializer** class code block, type the following code, press Spacebar, and then click, **Seed(OperasDB context)**.

```
override
```

21. In the **Seed** method, place the mouse cursor after the call to base.Seed, press Enter, and then type the following code.

```
var operas = new List<Opera>
{
   new Opera {
      Title = "Cosi Fan Tutte",
      Year = 1790,
      Composer = "Mozart"
   }
};
operas.ForEach(s =>
   context.Operas.Add(s));
context.SaveChanges();
```

22. On the **Build** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Build Solution**, and then note that the application is built successfully.
23. In the Solution Explorer pane, right-click **Controllers**, click **Add**, and then click **Controller**.
24. In the **Controller Name** box, type **OperaController**.
25. In the **Template** box, click **MVC controller with read/write actions and views, using Entity Framework**.
26. In the **Model Class** box, click **Opera (OperasWebSite.Models)**.
27. In the **Data context class** box, click **OpersDB (OperasWebSite.Models)**, and then click **Add**.
28. In the Solution Explorer pane, in the **Views/Operas** folder, double-click **Create.cshtml**.
29. In the Create.cshtml code window, locate and delete the following code.

```
@section Scripts {
   @Script.Render("~/bundles/jqueryval")
}
```

30. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual** Studio window, click **Start Debugging**

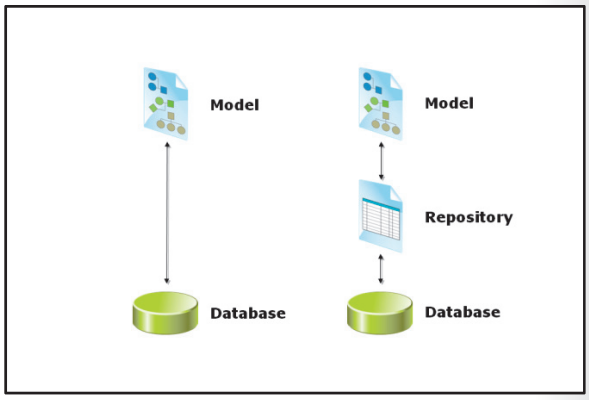
Note: The Internet Explorer window is displayed with an error message. The error message is expected because the home page view has not been added

31. In the Address bar of the Internet Explorer window, append the existing URL with **opera/index** and then click the **Go to** button.
32. On the Index page, click **Create New**.
33. In the **Title** box of the result page, type **Carmen**, and then, in the **Year** box, type **1475**.
34. In the **Composer** box, type **Bizet**, and then click **Create**.

Note: An error message is displayed by the custom validator.

35. In the **Year** box, type **1875**, and then click **Create**.
36. In the Internet Explorer window, click the **Close** button.
37. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Data Access in Models and Repositories



In MVC applications, you can place the data access code in the model, along with the business logic. However, many software architects prefer to separate these two types of code because they serve different purposes:

- *Business Logic*. This code defines the objects that the web application manages, their properties, and their relationships with each other.
- *Data Access Logic*. This code defines the operations necessary to persist data to a database. This includes operations to create new records, read records, update records, and delete records in database tables. A single object in the business logic layer may take data from multiple database tables. This abstraction is handled by the data access logic.

MVC does not require you to separate business and data access logic, and you can create MVC model classes that implement both layers. This is often done in small or simple web applications with small development teams. In these scenarios, Entity Framework classes are used directly in the model classes.

In more complex situations, you need to place the business logic in MVC model classes and place the data access logic in dedicated classes called repositories. When you take this approach, model classes are independent of the database structure and do not include code that depends on database table names, column names, or view names. This approach makes it easier to redesign the database or move to a different data store or data access technology, without the need to re-code your entire application. Using this approach, you employ the Entity Framework in your repository classes, but not in your model classes.

How to Separate Model Classes and Repositories

If you do choose to separate business logic and data access logic, you must take the following steps for each model class:

1. Define an interface for the repository class. This interface declares the methods that the repository class uses to read and write data from and to the database.
2. Create and write code for the repository class. This class must implement all the data access methods declared in the interface.
3. Remove all data access code from the model class.
4. Modify the controller class to use the repository class. Instead, create an instance of the repository class and call its methods to create the model.

A Simple Example Model Class and Repository Class

To illustrate how to separate model classes and repository classes, the following examples implement a comment on a photo.

The Comment model class helps users to comment on a photo, as the following lines of code illustrate.

The Comment Model Class

```
public class Comment
{
   public int CommentID { get; set; }
   public int PhotoID { get; set; }
   public string CommentText { get; set; }
   public virtual Photo Photo { get; set; }
}
```

This interface for a comment repository class defines just one method to get the comments for a given photo, as the following lines of code illustrate.

The ICommentRepository Interface

```
public interface ICommentRepository {
   ICollection<Comment> GetComments (int PhotoID);
}
```

The CommentRepository class implements the **GetComments** method as the following lines of code illustrate.

The CommentRepository Class

```
public class CommentRepository : ICommentRepository
{
   public ICollection<Comment> GetComments(int PhotoID)
   {
      //Implement entity framework calls here.
   }
}
```

The **CommentsController** class uses the repository class, instead of calling Entity Framework methods, as the following lines of code illustrate.

The CommentsController Class

```
public class CommentsController : Controller
{
   ICommentRepository commentRepository = new CommentRepository;
   public ActionResult DisplayCommentsForPhoto (int PhotoID)
   {
      //Use the repository to get the comments
      ICollection<Comments> comments = commentRepository.GetComments(PhotoID);
      return View("DisplayComments", comments);
   }
```

```
}
```

Note: By using the **ICommentRepository** interface, the code makes it easy to replace **CommentRespository** with another implementation if you need to. However, the **CommentController** code still creates a **CommentRespository** object. You have to modify the object to make the replacement.

In an even better architecture, you can replace **CommentRepository** with a different implementation of **ICommentRepository** without any changes to the **CommentController** class. This is an extremely flexible and adaptable approach and is called a loosely coupled architecture.

Loosely coupled architectures are also essential for unit testing.

Lab: Developing ASP.NET MVC 4 Models

Scenario

You are planning to create and code an MVC model that implements your plan for photos and comments in the Adventure Works photo sharing application. The model must store data in a Windows Azure SQL database and include properties that describe photos, comments, and their content. The model must enable the application to store uploaded photos, edit their properties, and delete them in response to user requests.

Objectives

After completing this lab, you will be able to:

- Create a new ASP.NET MVC 4 project in Visual Studio 2012.
- Add a new model to the ASP.NET MVC 4 web application and add properties to the model.
- Use display and edit data annotations in the MVC model to assign property attributes to views and controllers.
- Use Visual Studio to create a new Windows Azure SQL database and connect to the database.
- Add Entity Framework code to the model classes in the MVC model.
- Use display and edit data annotations in the MVC model to assign property attributes to views and controllers.

Estimated Time: 30 minutes

Virtual Machine: **20486B-SEA-DEV11**

Username: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Exercise 1: Creating an MVC Project and Adding a Model

Scenario

In this exercise, you will:

- Create a new MVC 4 web application in Visual Studio 2012.
- Add model classes to the web application.

The main tasks for this exercise are as follows:

1. Create a new MVC project.

2. Add a new MVC model.

▶ Task 1: Create a new MVC project.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Start Visual Studio 2012 and create a new ASP.NET MVC 4 web application by using the following information:

- Name: **PhotoSharingApplication**
- Location: **Allfiles (D):\Labfiles\Mod03**
- Solution name: **PhotoSharingApplication**
- Create directory for solution: True
- Project template: **Empty**

▶ Task 2: Add a new MVC model.

1. Add a new model class to the PhotoSharingApplication project by using the following information:
   - Class name: **Photo**
2. Add another model class to the PhotoSharingApplication project by using the following information:
   - Class name: **Comment**

Results: After completing this exercise, you will be able to create an MVC 4 web application and add model classes to the web application.

Exercise 2: Adding Properties to MVC Models

Scenario

In this exercise, you will:

- Add properties to the Photo and the Comment model classes.
- Implement a relationship between model classes.

The main tasks for this exercise are as follows:

1. Add properties to the Photo model class.
2. Add properties to the Comment model class.
3. Implement a relationship between model classes.

▶ Task 1: Add properties to the Photo model class.

1. Add a primary key property to the Photo model class by using the following information:
   - Scope: **public**
   - Property name: **PhotoID**
   - Data type: **integer**
   - Access: **Read and write**
2. Add a title property to the Photo model class by using the following information:
   - Scope: **public**
   - Property name: **Title**
   - Data type: **string**
   - Access: **Read and write**
3. Add an image property to the Photo model class and store the MIME type of image by using the following information:

- Scope: **public**
- Property names: **PhotoFile**, **ImageMimeType**
- Data type for the image: **byte []**
- Data type for MIME type: **string**
- Access: **Read and write**

4. Add a description property to the Photo model class by using the following information:
   - Scope: **public**
   - Property name: **Description**
   - Data type: **String**
   - Access: **Read and write**
5. Add a date property to the Photo model class by using the following information:
   - Scope: **public**
   - Property name: **CreatedDate**
   - Data type: **DateTime**
   - Access: **Read and write**
6. Add a user name property to the Photo model class by using the following information:
   - Scope: **public**
   - Property name: **UserName**
   - Data type: **string**
   - Access: **Read and write**

▶ Task 2: Add properties to the Comment model class.

1. Add a primary key to the Comment model class by using the following information:
   - Scope: **public**
   - Property name: **CommentID**
   - Data type: **integer**
   - Access: **Read and write**
2. Add a PhotoID property to the Comment model class by using the following information:
   - Scope: **public**
   - Property name: **PhotoID**
   - Data type: **integer**
   - Access: **Read and write**
3. Add a user name property to the Comment model class by using the following information:
   - Scope: **public**
   - Property name: **UserName**
   - Data type: **string**
   - Access: **Read and write**

4. Add a subject property to the Comment model class by using the following information:
   - Scope: **public**
   - Property name: **Subject**
   - Data type: **string**
   - Access: **Read and write**
5. Add a body text property to the Comment model class by using the following information:
   - Scope: **public**
   - Property name: **Body**
   - Data type: **string**
   - Access: **Read and write**

▶ Task 3: Implement a relationship between model classes.

1. Add a new property to the Photo model class to retrieve comments for a given photo by using the following information:
   - Scope: **public**
   - Property name: **Comments**
   - Data type: a collection of **Comments**
   - Access**: Read and write**
   - Include the **virtual** keyword
2. Add a new property to the Comment model class to retrieve the photo for a given comment by using the following information:
   - Scope: **public**
   - Property name: **Photo**
   - Property type: **Photo**
   - Access: **Read and write**
   - Include the **virtual** keyword

Results: After completing this exercise, you will be able to add properties to classes to describe them to the MVC runtime. You will also implement a one-to-many relationship between classes.

Exercise 3: Using Data Annotations in MVC Models

Scenario

In this exercise, you will:

- Add data annotations to the properties to help MVC web application render them in views and validate user input.

The main tasks for this exercise are as follows:

1. Add display and edit data annotations to the model.
2. Add validation data annotations to the model.

▶ Task 1: Add display and edit data annotations to the model.

1. Add a display data annotation to the Photo model class to ensure that the PhotoFile property is displayed with the name, Picture.
2. Add an edit data annotation to the Photo model class that ensures the Description property editor is a multiline text box.
3. Add the following data annotations to the Photo model class to describe the CreatedDate property:
   - Data type: **DateTime**
   - Display name: **Created Date**
   - Display format: **{0:MM/dd/yy}**
4. Add an edit data annotation to the Comment model class that ensures that the Body property editor is a multiline text box.

▶ Task 2: Add validation data annotations to the model.

1. Add a validation data annotation to the Photo model class to ensure that the users complete the Title field.
2. Add validation data annotations to the Comment model class to ensure that the users complete the Subject field and enter a string with a length shorter than 250 characters.

Results: After completing this exercise, you will be able to add property descriptions and data annotations to the two model classes in the MVC web application.

Exercise 4: Creating a New Windows Azure SQL Database

Scenario

In this exercise, you will:

- Add Entity Framework code to the Photo Sharing application in code-first mode.
- Create a new SQL database in Windows Azure.
- Use the SQL database to create a connection string in the application.

The main tasks for this exercise are as follows:

1. Add an Entity Framework Context to the model.
2. Add an Entity Framework Initializer.
3. Create a Windows Azure SQL database and obtain a connection string.

▶ Task 1: Add an Entity Framework Context to the model.

1. Use the NuGet Package Manager to add Entity Framework 5.0 to the application.
2. Add a new class called PhotoSharingContext to the Models folder and ensure that the new class inherits the System.Data.Entity.DbContext class.
3. Add public **DbSet** properties to Photos and Comments to enable Entity Framework to create database tables called Photos and Comments.

▶ Task 2: Add an Entity Framework Initializer.

1. Add a new class called PhotoSharingInitializer to the Models folder and ensure that the new class inherits the DropCreateDatabaseAlways <PhotoSharingContext> class.
2. Open the getFileBytes.txt file from the following location and add all the text of the file as a new method to the PhotoSharingInitializer class:
   - File path: **Allfiles (D):\Labfiles\Mod03\CodeSnippets**
3. Override the **Seed** method in the **PhotoSharingInitializer** class.
4. Create a new list of **Photo** objects in the **Seed** method. The list should contain one photo object with the following properties:
   - Title: **Test Photo**
   - Description: *<A description of your choice>*
   - UserName: **NaokiSato**
   - PhotoFile: **getFileBytes("\\Images\\flower.jpg")**
   - ImageMimeType: **image/jpeg**
   - CreatedDate: *<Today's date>*
5. Add each **photo** object in the **photos** list to the Entity Framework context and then save the changes to the context.
6. Create a new list of Comment objects in the **Seed** method. The list should contain one **Comment** object with the following properties:
   - PhotoID: **1**
   - UserName: **NaokiSato**
   - Subject: **Test Comment**
   - Body: **This comment should appear in photo 1**
7. Add the comment list to the Entity Framework context and save the comment to the database.
8. Open **Global.asax** and add a line of code to the **Application_Start** method that calls **Database.SetInitializer**, passing a new **PhotoSharingInitializer** object. Also add the following namespaces:
   - **using System.Data.Entity;**
   - **using PhotoSharingApplication.Models;**

▶ Task 3: Create a Windows Azure SQL database and obtain a connection string.

1. Log on to the Windows Azure portal by using the following information:
   - Portal address: **http://www.windowsazure.com/**
   - User name: *<your Windows Live user name>*
   - Password: *<your Windows Live password>*
2. Create a new database server and a new database by using the following information:
   - Database name: **PhotoSharingDB**
   - Database server: **New SQL Database Server**

- Login name: <*your first name*>
- Login password: **Pa$$w0rd**
- Login password confirmation: **Pa$$w0rd**
- Region: <*a region close to you*>

3. In the list of allowed IP addresses for the **PhotoSharingDB** database, add the following IP address ranges:
   - Rule name: **First Address Range**
   - Start IP Address: <*first address in range*>
   - End IP Address: <*last address in range*>
4. Obtain the connection string for the PhotoSharingDB database and add it to the Web.config file.
5. Build the Photo Sharing application.

Results: After completing this exercise, you will be able to create an MVC application that uses Windows Azure SQL Database as its data store.

Exercise 5: Testing the Model and Database

Scenario

In this exercise, you will:

- Add a controller and views to the MVC web application.
- Run the web application.

The main tasks for this exercise are as follows:

1. Add a controller and views.

2. Add an image and run the application.

▶ Task 1: Add a controller and views.

1. Add a new controller to the **PhotoSharingApplication** project by using the following information:
   - Name**: PhotoController**
   - Template: **MVC Controller with read/write actions and views, using Entity Framework**
   - Model class: **Photo**
   - Data context class**: PhotoSharingContext**
   - Views: **Razor(CSHTML)**

▶ Task 2: Add an image and run the application.

1. Create a new top-level folder, and copy an image to the new folder by using the following information:
   - New folder name: **Images**
   - Image to be copied: **flower.JPG**
   - Location of the image: **Allfiles (D):\Labfiles\Mod03\Images**
2. Run the application by debugging, and access the following relative path:

- **/photo/index**

Results: After completing this exercise, you will be able to add controllers, views, and images to an MVC web application and test the application by displaying data from a Windows Azure SQL database.

Question: You are building a site that collects information from customers for their accounts. You want to ensure that customers enter a valid email address in the **Email** property. How would you do this?

Question: You have been asked to create an intranet site that publishes a customer database, created by the sales department, to all employees within your company. How would you create the model with Entity Framework?

Module Review and Takeaways

The heart of an MVC web application is the model. This is because the model classes describe the information and objects that your web application manages. In this module, you have seen how to create your model, define relationships between the model classes, describe how to display, edit, and validate properties, and how to extend MVC model handling capabilities. You have also seen how to bind model classes to database tables by using Entity Framework and how to query object in the model by writing LINQ code.

Best Practice: If you have a pre-existing database for a web application, use Entity Framework in the database-first workflow to import and create your model and its classes.

Best Practice: If you want to create a new database for a web application and prefer to draw your model in a Visual Studio designer, use Entity Framework in the model-first workflow to create your model and its classes.

Best Practice: If you want to create a new database for a web application and prefer to write code that describes your model, use Entity Framework in the code-first workflow to create your model and its classes.

Best Practice: If you want to separate business logic from data access logic, create separate model classes and repository classes.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The website cannot connect to or create a database.	

Review Question(s)

Question: At the end of the first iteration of your project, you have a website that displays photos that users upload. However, during development, the database is empty and users must upload several photos to the site so they can test the functionality. Your manager wants you find some way to populate the database whenever it is deployed to the test server. How can you do this?

Module 04

Developing ASP.NET MVC 4 Controllers

Contents:

Module Overview

MVC controllers respond to browser requests, create model objects, and pass them to views for rendering and displaying in the web browser. If required, controllers can also perform other actions, such as saving model class changes to the database. Controllers are central to MVC applications. You need to understand the functioning of controllers to be able to create the right model objects, manipulate them, and pass them to the right views.

To maximize the re-use of code in controllers, you must know how to program action filters. You can use action filters to run code before or after every action in your web application, on every action in a controller, or on other combinations of controller actions.

Objectives

After completing this module, you will be able to:

- Add a controller to a web application that responds to user actions specified in the project design.
- Write code in action filters that runs before or after a controller action.

Lesson 1
Writing Controllers and Actions

A controller is a .NET Framework class that inherits from the **System.Web.Mvc.Controller** base class. Controllers respond to user requests. Within a controller class, you create actions to respond to user requests. Actions are methods within a controller that return an **ActionResult** object. The **ActionResult** object is often a view that displays a response to the user request; however, it can also yield other types of results. To process incoming user requests, manage user input and interactions, and implement relevant application logic, you need to know how to create controllers and actions. You must also know how to create parameters in action code blocks and pass objects to actions.

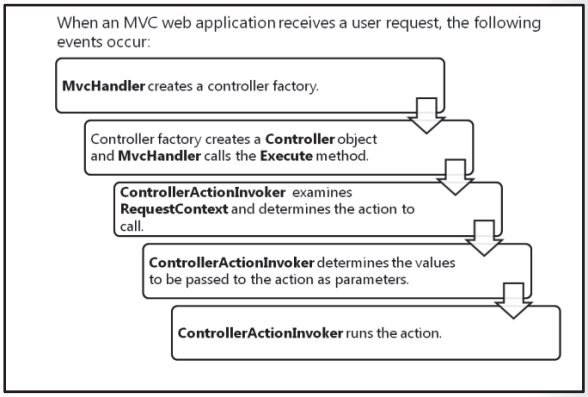
Lesson Objectives

After completing this lesson, you will be able to:

- Describe how a controller responds to user actions in an MVC 4 web application.
- Write controller actions to respond to web browser requests, create model classes, and call views.
- Explain how to use parameters passed in a browser request to a controller action and use them to change the action result.
- Explain how to pass information to views that have model classes.
- Create a controller and actions.
- Describe controller factories.

Responding to User Requests

When an MVC web application receives a request from a web browser, the following events happen in sequence:



1. An **MvcHandler** object creates a controller factory. The controller factory is the object that instantiates a controller to respond to the request. Usually, this factory is a **DefaultControllerFactory** object, but you can create a custom controller factory, if necessary. The **MvcHandler** object chooses the controller factory based on the **RequestContext** object, which has information about the request that the user made.
2. The controller factory creates a **Controller** object, and the **MvcHandler** calls the **Execute** method in that controller.
3. The **ControllerActionInvoker** examines the **RequestContext** object and determines the action to call in the **Controller** object.
4. The **ControllerActionInvoker** uses a model binder to determine the values to be passed to the action as parameters.
5. The **ControllerActionInvoker** runs the action. Often, the action creates a new instance of a model class, perhaps by querying the database with the parameters that the invoker passed to it. This model object is passed to a view, to display results to the user. Action methods can do many other things

such as rendering views and partial views, redirecting to other websites, displaying snippets of content, or displaying other files.

Note: The routing engine determines which controller and action receives a request. Routing is not covered in this section.

The User Request

Users of web browsers make requests either by typing a URL into the Address bar of the browser, or by clicking a link to some address within your website. Such links can either be within your website, in which case you can control how they are rendered, or from an external website. Whether the request originates from within the website or from an external website, it can include information that controller actions can use as parameters. Consider the following examples:

- http://www.adventureworks.com/: This URL is the home page of the website and specifies no further information.
- http://www.adventureworks.com/photo: This URL specifies an extra value, **photo**. By default, the **MvcHandler** interprets this as the name of a controller.
- http://www.adventureworks.com/photo/index: This URL specifies a second value, **index**. By default, the **MvcHandler** interprets this as the name of an action within the controller.
- http://www.adventureworks.com/photo/display/1: This URL specifies a third value, **1**. By default, the **ControllerActionInvoker** interprets this as a parameter to pass to the action method.
- http://www.adventureworks.com/photo/display?id=1: This URL includes a query string, **id=1**. The model binder examines the **Display** actions in the **Photo** controller. If it finds an action with a parameter called **id**, it calls that action and passes **1** as a parameter.

Note: You can modify the preceding logic in several ways. For example, you can create routes that interpret the preceding URLs differently. The examples are true when only the default route exists.

Let us consider that a user requests a controller called **photo**, by typing the URL in the Address bar of a web browser. By default, the MVC **DefaultControllerFactory**, names this controller class as **PhotoController**. You should keep to this convention when you create and name controllers. Otherwise, you will receive unexpected 404 errors and controllers will not work as intended. If you create a custom controller factory, you can define your own naming convention for controller classes.

The Microsoft Visual Studio project templates include a folder named **Controllers**. This is a good location to create your controllers. Microsoft Visual Studio places controllers in the ***projectname*.Controllers** namespace, by default.

The following example shows the code in an MVC controller class called, **PhotoController**. The controller has a single action called **Index**, which returns a list of **Photo** items from the Entity Framework context.

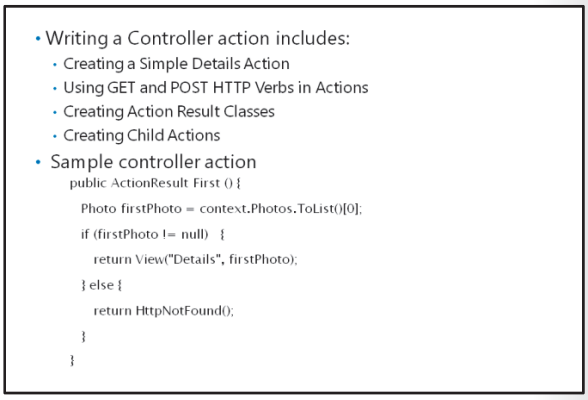
A Simple Controller Class

```
public class PhotoController : Controller
{
   private ContextDB context = new ContextDB();
   public ActionResult Index ()
   {
      return View("Index", context.Photos.ToList());
   }
}
```

Question: What is the convention that you should follow while creating controllers?

Writing Controller Actions

Controllers encapsulate user interaction logic for an MVC web application. You specify this logic by writing actions. An action is a method within the controller class. The code you write within an action method determines how the controller responds to the request, and the model class and view that MVC uses to display a webpage in the browser.



Note: When you add a new controller to an MVC application, Visual Studio presents scaffolding options to help you create action methods and associated views. For example, if you specify a model class and Entity Framework context class, Visual Studio can create scaffold index and details, and create, edit, and delete action methods in your new controller. You can use these as a starting point for your code. As you become more experienced with action methods, you may prefer to select the **Empty MVC Controller** template and write your own methods without scaffold code.

Controller actions are public methods that return an **ActionResult** object. Alternatively, actions can return objects of many other classes that derive from the **ActionResult** class. For example, you can write code for a controller with an **Index** action to obtain all **Photo** objects and pass them to the **Index** view.

When you want to display a specific model object, you must obtain the correct instance from the database. The following code shows how to display the first **Photo** object.

A Details Action

```
public ActionResult First ()
{
   Photo firstPhoto = context.Photos.ToList()[0];
   if (firstPhoto != null)
   {
      return View("Details", firstPhoto);
   }
   else
   {
      return HttpNotFound();
   }
}
```

Some user actions are in two parts. For example, to create a new **Photo**, a user can make an HTTP GET request to the following URL: http://www.adventureworks.com/photo/create.

The following code shows an action that responds to a GET request, to display a new photo form.

A Create Action for the GET Request

```
public ActionResult Create ()
{
   Photo newPhoto = new Photo();
   return View("Create", newPhoto)
}
```

The **Create** view displays a form where users can fill photo details, such as the title, description, and so on. When a user clicks the **Submit** button, the web browser makes an HTTP POST request.

The following action method responds to a POST request. Note that the method name is the same, but the [HttpPost] annotation is used to specify that this action responds to the HTTP POST verb.

A Create Action for the POST Request

```
 [HttpPost]
public ActionResult Create (Photo photo)
{
   if (ModelState.IsValid)
   {
      context.Photos.Add(photo);
      context.SaveChanges();
      return RedirectToAction("Index");
   }
   else
   {
      return View("Create", photo);
   }
}
```

Note that the **ModelState.IsValid** property is used to check whether the user has submitted valid data. You can specify data validation by using validation data annotations in the model class. If the data is valid, the model object is added and saved. Otherwise, the application displays the **Create** view again, so that the user can correct the invalid data.

Note: For the preceding code to work, the **Create** view must contain a form that uses the **POST** method.

Possible Return Classes

Action methods are usually defined with the **ActionResult** class as the return type. **ActionResult** is a base class, and you can use a range of derived classes to return different responses to the web browser.

Controller actions usually return a view and pass a model class to it, for display. You can create an action that calls the **View()** helper, and creates and returns a **ViewResult** object. The **View()** helper is available when you derive from the base **Controller** class.

Alternatively, you can return an HTTP error. For example, you can create an action such that if a **Photo** object is not found, the code creates a 404 not found error by using the **HttpNotFound()** helper.

Sometimes, you may want to return a file from an action method. For example, let us consider that in a **Photo** model, the image file is stored as a byte array in the database. To display this byte array as an image on a webpage, the action must return it as a .jpeg file, which can be used for the **src** attribute of an **<img>** HTML tag. To return files, you can use the **File()** helper to return a **FileContentResult** object. You can use this technique in the **GetImage** action.

Other possible action results include:

- *PartialViewResult*. You can use this action to generate a section of an HTML page, but not a complete HTML page. Partial views can be re-used in many views throughout a web application.
- *RedirectToRouteResult*. You can use this action result to redirect the web browser to another action method or another route.
- *RedirectResult*. You can use this action result to redirect to a specific URL, either inside your web application or to an external location.

- *ContentResult*. You can use this action result to return text to the web browser. You can return plain text, XML, a comma-separated table, or other text formats. This text can be rendered in the web browser or parsed with client-side code.

Child Actions

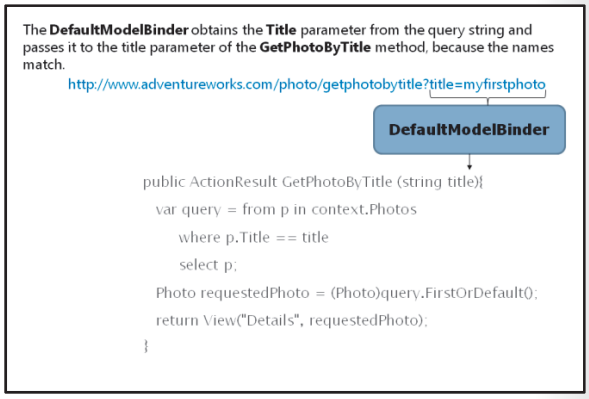
When an action returns a complete view, MVC sends a new complete webpage to the web browser for display. Sometimes, you may want to call an action from within a view, to return a piece of content for display within a webpage. A child action is an action method that can return a small piece of content in this manner. The **FileContentResult** is often a good example of a child action, because the image returned usually forms part of a webpage. Partial views also support child actions.

To declare an action method as a child action, you can use the **[ChildActionOnly]** annotation. This annotation ensures that the action method can be called only from within a view by using the **Html.Action()** helper. Using this method, you can prevent a user from calling the child action directly by typing the correct URL into the Address bar.

Question: What are the various **ActionResult** return types that you can write as code while creating a controller?

Using Parameters

When users request webpages, they often specify information other than the name of the webpage itself. For example, when they request a product details page, they may specify the name or catalog number of the product to display. Such extra information is referred to as parameters. You must understand how to determine in code what parameters the user sent in their request.



The **ControllerActionInvoker** and the **DefaultModelBinder** classes obtain parameters from a user request and pass them to action methods. The **DefaultModelBinder** can locate parameters in a posted form, the routing values, the query string, or in the posted files. If the model binder finds a parameter in the action method that matches the name and type of a parameter from the request, the action method is called and the parameter is passed from the request. This arrangement enables you to obtain and use parameters in your actions. For example, if a user requests the URL http://www.adventureworks.com/photo/getphotobytitle/?title=myfirstphoto, you can easily obtain title values in your action method.

The following example code shows how to determine the value of the title parameter in a controller action.

Using a Query String Parameter

```
public ActionResult GetPhotoByTitle (string title)
{
   var query = from p in context.Photos
        where p.Title == title
        select p;
   Photo requestedPhoto = (Photo)query.FirstOrDefault();
   if (requestedPhoto != null)
   {
      return View("Details", requestedPhoto);
   }
```

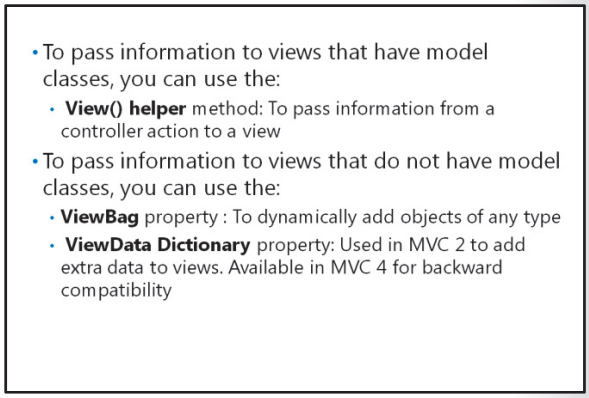
```
    else
    {
        return HttpNotFound();
    }
}
```

Note that the action method code uses the parameter **title** to formulate a LINQ to Entities query. In this case, the query searches for a **Photo** with the specified **Title**. Parameters in action methods are frequently used in this manner.

Note: The example works if the **DefaultModelBinder** passes parameters. If you create a custom model binder in your application, you must ensure that it passes parameters in the correct manner. Otherwise, the action method cannot access the parameters that the user specified in the request.

Question: How does **DefaultModelBinder** pass parameters?

Passing Information to Views



You can pass a model object, such as a **Photo**, from the action method to the view by using the **View()** helper method. This is a frequently-used method to pass information from a controller action to a view. This is because this method adheres closely to the Model-View-Controller pattern, in which each view renders the properties found in the model class, which the view receives from the controller. You should use this approach wherever possible.

However, in some cases, you may want to augment the information in the model class with some extra values. For example, you may want to send a title, which needs to be inserted in the page header, to the view. Furthermore, some views do not use model classes. The home page of a website, for example, often does not have a specific model class. To help in these situations, you can use two other methods to provide extra data: **ViewBag** and **ViewData**.

Using The ViewBag

The **ViewBag** is a dynamic object that is part of the base controller class. Because it is a dynamic object, you can add properties that are of any type to it, in the action method. In the view, you can use the **ViewBag** object to obtain the values added in the action.

You can add properties to the **ViewBag** object in the action method as illustrated in the following lines of code.

Adding Properties to the ViewBag Object

```
ViewBag.Message = "This text is not in the model object";
ViewBag.ServerTime = DateTime.Now;
```

To obtain and use the same properties in a view, you can use the following Razor code.

Using ViewBag Properties

```
<p>
   The message of the day is: @ViewBag.Message
</p>
<p>
   The time on the server is: @ViewBag.ServerTime.ToString()
</p>
```

Using The ViewData Dictionary

The **ViewBag** object was added to MVC in version 3. In the earlier versions, you could pass extra data to views by using the **ViewData** dictionary. This feature is still available in MVC 4 for backward compatibility and for developers who prefer to use dictionary objects. In fact, **ViewBag** is a dynamic wrapper above the **ViewData** dictionary. This means that you could save a value in a controller action by using **ViewBag** and read the same value back out by using **ViewData**.

In action methods, you can add data to the **ViewData** dictionary by using key/value pairs as the following lines of code illustrate.

Adding Data to the ViewData

```
ViewData["Message"] = " This text is not in the model object"
ViewData["ServerTime"] = DateTime.Now;
```

To obtain and use the same values in a view, you can use the following Razor code.

Using ViewData Values

```
<p>
   The message of the day is: @ViewData["Message"]
</p>
<p>
   The time on the server is: @((DateTime)ViewData["ServerTime"])
</p>
```

In the examples, note that you can cast any **ViewData** values other than strings.

Question: Do **ViewBag** and **ViewData** serve different purposes?

Demonstration: How to Create a Controller

In this demonstration, you will see how to create a controller and write common actions in the controller.

Note: At the end of this demonstration, the application will not include views. Therefore, the application cannot display webpages.

Demonstration Steps

1. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, right-click **Controllers**, point to **Add**, and then click **Controller**.
2. In the **Controller Name** box of the **Add Controller** dialog box, type **OperaController**.
3. In the **Template** box, click **Empty MVC controller**, and then click **Add**.
4. In the OperaController.cs code window, locate the following code.

```
using System.Web.MVC;
```

5. Place the mouse cursor at the end of the System.Web.MVC namespace, press Enter, and then type the following code.

```
using System.Data.Entity;
using OperasWebSite.Models;
```

6. In the **OperaController** class code block, press Enter, type the following code, and then press Enter.

```
private OperasDB contextDB =
   new OperasDB();
```

7. In the **Index** action code block, select the following code.

```
   return View();
```

8. Replace the selected code with the following code.

```
return View("Index",
   contextDB.Operas.ToList());
```

9. Place the mouse cursor at the end of the **Index** action code block, press Enter, and then type the following code.

```
public ActionResult Details (int id)
{
}
```

10. In the **Details** action code block, type the following code.

```
Opera opera =
   contextDB.Operas.Find(id);
if (opera != null)
{
   return View("Details", opera);
}
else
{
   return HttpNotFound();
}
```

11. Place the mouse cursor at the end of the **Details** action code block, press Enter twice, and then type the following code.

```
public ActionResult Create ()
{
}
```

12. In the **Create** action code block, type the following code.

```
Opera newOpera = new Opera();
return View("Create", newOpera);
```

13. Place the mouse cursor at the end of the **Create** action code block, press Enter twice, and then type the following code.

```
[HttpPost]
public ActionResult Create
   (Opera newOpera)
{
}
```

14. Place the mouse cursor in the **Create** action code block with the HTTP verb **POST**, and then type the following code.

```
if (ModelState.IsValid)
{
   contextDB.Operas.Add(newOpera);
   contextDB.SaveChanges();
return
      RedirectToAction("Index");
}
else
{
   return View("Create", newOpera);
}
```

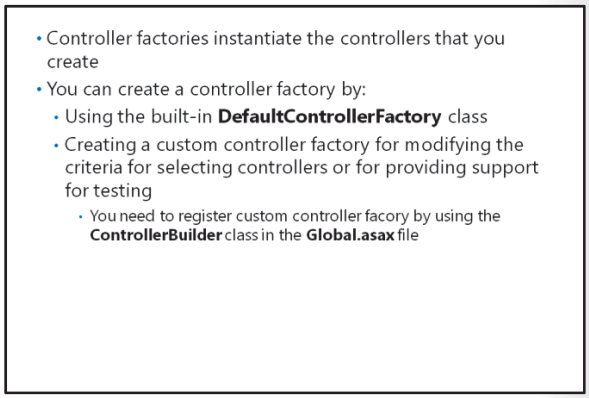
15. On the **FILE** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Save Controllers\OperaControllers.cs**.
16. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Note: The message, "Save changes to the following items?" is displayed.

17. In the Microsoft Visual Studio dialog box, note that the message, "Save changes to the following items?" is displayed, and then click Yes.

What Are Controller Factories?

A controller factory is an MVC component that instantiates the controller classes that you create. For example, when a user requests a list of **Photo** model objects, a controller factory should create an instance of the **PhotoController** class. An action invoker then calls one of the action methods in that class, and a model binder passes parameters to it.



The MVC framework includes a built-in **DefaultControllerFactory** class that is suitable for most web applications. However, you must understand how **DefaultControllerFactory** determines the controller class that it needs to create. Occasionally, you may need to create a custom controller factory to implement your own controller creation logic.

How the DefaultControllerFactory Class Locates a Controller Class

The **DefaultControllerFactory** class identifies controller classes by using the following criteria:

- The class scope must be **public**.
- The class must not be marked as **abstract**.
- The class must not take generic parameters.
- The class must have a name that ends with **Controller**.
- The class must implement the **IController** interface.

When the MVC web application starts, **DefaultControllerFactory** creates a list of all the classes in the application that satisfy these criteria. This list helps to create the correct controller rapidly. To write a controller, you must ensure that all the above mentioned criteria are implemented. Usually, you meet the **IController** interface criterion by inheriting from the base **Controller** class.

By default, the DefaultControllerFactory mandates all controller classes to end with the word **Controller**. For example, following this convention, for the **Photo** model class, you would create a controller called **PhotoController**.

Creating a Custom Controller Factory

Occasionally, you might want to implement a custom controller factory. There are two common reasons for doing this:

- *To modify the criteria for selecting controllers*. The criteria described earlier are suitable for most web applications, but sometimes, you may want to change them. For example, you may not want to name controllers with **Controller** at the end, or you may want to add extra criteria of your own.
- *To support direct injection for testing*. Direct injection is a programming technique that lets you specify classes at run time, instead of specifying classes when writing code. This is helpful for unit testing because you can inject a test class with mock data, instead of real data. The **DefaultControllerFactory** class does not support direct injection.

The following code shows how to create a custom controller factory by implementing the **IControllerFactory** interface.

A Custom Controller Factory

```
public class AdWorksControllerFactory : IControllerFactory
{
   public IController CreateController (RequestContext requestContext, string
ControllerName)
   {
      Type targetType = null;
      if (ControllerName == "Photo")
      {
         targetType = typeof(PhotoController);
      }
      else
      {
         targetType = typeof(GeneralPurposeController);
      }
      return targetType == null ? null :
(IController)Activator.CreateInstance(targetType);
   }
   public SessionStateBehavior GetControllerSessionBehavior(RequestContext
requestContext,
      string controllerName)
   {
      return SessionStateBehavior.Default;
   }
   public void ReleaseController(IController controller)
   {
      IDisposable disposable = controller as IDisposable;
      if (disposable != null)
      {
         disposable.Dispose();
      }
   }
}
```

You must implement the **CreateController**, **GetControllerSessionBehavior**, and **ReleaseController** methods for any custom controller factory you create.

For example, if the controller name passed to a controller factory is "Photo", then the **PhotoController** is used. Otherwise, the **GeneralPurposeController** is used. The logic in a real custom controller should be more sophisticated than in this example. However the example illustrates the minimal required code to create a custom controller factory.

Registering a Custom Controller Factory

Even if you create a custom controller factory in your application, MVC will still use the **DefaultControllerFactory** class, unless you register your custom factory.

You register a custom controller factory by using the **ControllerBuilder** class in the **Global.asax** file, as the following lines of code show.

Registering a Custom Controller Factory

```
protected void Application_Start()
{
   ControllerBuilder.Current.SetControllerFactory(new AdWorksControllerFactory());
}
```

Question: Can you create a controller that does not end with "Controller"?

Lesson 2
Writing Action Filters

In some situations, you may need to run code before or after controller actions run. Before a user runs any action that modifies data, you might want to run the code that checks the details of the user account. If you add such code to the actions themselves, you will have to duplicate the code in all the actions where you want the code to run. Action filters provide a convenient way to avoid code duplication. You need to know how to create and use action filters in your web application, and when to use them.

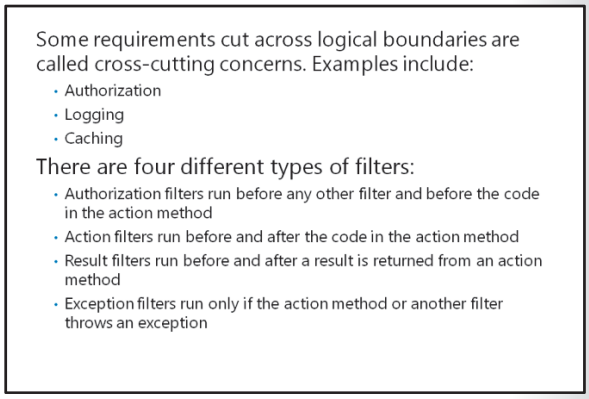
Lesson Objectives

After completing this lesson, you will be able to:

- Describe action filters.
- Create action filters.
- Determine when to use action filters.

What Are Filters?

The MVC programming model enforces the separation of concerns. For example, the business logic in model classes is separate from the input logic in controllers and the user interface logic in views. Each model class is also clearly distinct from other model classes. However, there are scenarios where requirements may be relevant to many parts of your application and cut across logical boundaries. For example, authorization must be done for many sensitive actions and controllers, regardless of the model and views that they return. These types of requirements are known as cross-cutting concerns. Some common examples of cross-cutting concerns include authorization, logging, and caching.



Filters

Filters are MVC classes that you can use to manage cross-cutting concerns in your web application. You can apply a filter to a controller action by annotating the action method with the appropriate attribute. For example, an action annotated with the **[Authorize]** attribute, can be run only by authenticated users. You can also apply a filter to every action in a controller by annotating the controller class with the attribute.

Filter Types

There are four types of filters that you can use in MVC. These filters run at slightly different stages in the request process.

Filter Type	Interface	Default Class	Description
Authorization	IAuthorizationFilter	AuthorizeAttribute	Runs before any other filter and before the code in the action method. Used to check a user's access rights for the action.

Filter Type	Interface	Default Class	Description
Action	IActionFilter	ActionFilterAttribute	Runs before and after the code in the action method.
Result	IResultFilter	ActionFilterAttribute	Runs before and after a result is returned from an action method.
Exception	IExceptionFilter	HandleErrorAttribute	Runs only if the action method or another filter throws an exception. Used to handle errors.

Question: Which filter type will you use for the following actions?

1. Intercepting an error
2. Modifying a result
3. Authorizing users
4. Inspecting a returned value

Creating and Using Action Filters

If you have a cross-cutting concern in your web application, you can implement it by creating a custom action filter or a custom result filter. You can create custom filters by implementing the **IActionFilter** interface or the **IResultFilter** interface. However, the **ActionFilterAttribute** base class implements both the **IActionFilter** and I**ResultFilter** interfaces for you. By deriving your filter from the **ActionFilterAttribute** class, you can create a single filter that can run code both before and after the action runs, and both before and after the result is returned.

Sample Action Filter

```
public class SimpleActionFilter : ActionFilterAttribute {
  public override void OnActionExecuting
    (ActionExecutingContext filterContext) {
    Debug.WriteLine("This Event Fired: OnActionExecuting");
  }
  public override void OnActionExecuted
    (ActionExecutedContext filterContext) {
    Debug.WriteLine("This Event Fired: OnActionExecuted");
  }
}
```

The following code shows how an action filter is used to write text to the Visual Studio Output window in the order in which the IActionFilter and IResultFilter events run. Place this code in a class file within your web application.

A Simple Custom Action Filter

```
public class SimpleActionFilter : ActionFilterAttribute
{
   public override void OnActionExecuting(ActionExecutingContext filterContext)
   {
      Debug.WriteLine("This Event Fired: OnActionExecuting");
   }
   public override void OnActionExecuted(ActionExecutedContext filterContext)
   {
      Debug.WriteLine("This Event Fired: OnActionExecuted");
   }
   public override void OnResultExecuting(ResultExecutingContext filterContext)
   {
      Debug.WriteLine("This Event Fired: OnResultExecuting");
   }
   public override void OnResultExecuted(ResultExecutedContext filterContext)
   {
      Debug.WriteLine("This Event Fired: OnResultExecuted");
```

```
    }
}
```

Note: You can also create a custom authorization filter by implementing the **IAuthorizationFilter** interface. However, the default **AuthorizeAttribute** implementation is highly useful and satisfies almost all authentication requirements. You should be careful when overriding the default security code in MVC or any other programming model. If you do so without a full understanding of the implications, you can introduce security vulnerabilities that a malicious user can exploit.

Using a Custom Action Filter

After you have created a custom action filter, you can apply it to any action method or class in your web application by annotating the method or class with the action filter name.

In the following lines of code, the **SimpleActionFilter** is applied to the **Index** action of the **Photo** controller.

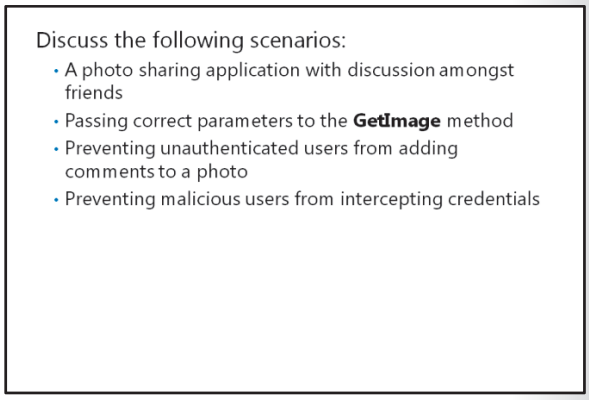
Using A Custom Action Filter

```
public class PhotoController : Controller
{
   ContextDB contextDB = new ContextDB();
   [SimpleActionFilter]
   public ActionResult Index()
   {
      return View("Index", contextDB.Photos.ToList());
   }
}
```

Question: What are the advantages of custom action filters?

Discussion: Action Filter Scenarios

Consider the following scenarios. In each case, discuss with the rest of the class whether the scenario requires a custom filter, or can be solved with a built-in filter type, or cannot be solved with filters.



1. You are writing a photo sharing application and you want to enable each user to discuss photos, cameras, lenses, and other photography equipment with other users whom they have marked as their friends. Other users should be prevented from seeing these discussions.
2. You want to ensure that when MVC calls the **GetImage** action method, the ID in the query string is passed as a parameter.
3. You are writing a photo sharing application and you want to prevent unauthenticated users from adding comments to a photo.
4. You want to prevent malicious users from intercepting the credentials entered by users in the logon form for your web application. You want to ensure that the credentials are encrypted.

Lab: Developing ASP.NET MVC 4 Controllers

Scenario

You have been asked to add a controller to the photo sharing application that corresponds to the Photo model class that you have created in an earlier module. The controller should include actions that respond when users upload photos, list all photos, display a single photo, and delete photos from the application. You should also add an action that returns the photo as a .jpg file to show on a webpage.

The members of your development team are new to ASP.NET MVC and they find the use of controller actions confusing. Therefore, you need to help them by adding a component that displays action parameters in the Visual Studio Output window whenever an action runs. You will add an action filter to achieve this.

Note: The controllers and views that you have added in Lab 2 were to test your new model classes. They have been removed from the project to create the actual controllers. You will create temporary views to test these controllers at the end of this lab.

Objectives

After completing this lab, you will be able to:

- Add an MVC controller to a web application.
- Write actions in an MVC controller that respond to user operations such as create, index, display, and delete.
- Write action filters that run code for multiple actions.

Lab Setup

Estimated Time: 60 minutes

Virtual Machine: **20486B-SEA-DEV11**

User name: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

- On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.
- In the navigation pane of the **Options** dialog box, click **Package Manager**.
- Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Adding an MVC Controller and Writing the Actions

Scenario

In this exercise, you will create the MVC controller that handles photo operations. You will also add the following actions:

- *Index*. This action gets a list of all the Photo objects and passes the list to the Index view for display.
- *Display*. This action takes an ID to find a single Photo object. It passes the Photo to the Display view for display.

- *Create (GET)*. This action creates a new Photo object and passes it to the Create view, which displays a form that the visitor can use to upload a photo and describe it.
- *Create (POST)*. This action receives a Photo object from the Create view and saves the details to the database.
- *Delete (GET)*. This action displays a Photo object and requests confirmation from the user to delete the Photo object.
- *DeleteConfirmed (POST)*. This action deletes a Photo object after confirmation.
- *GetImage:* This action returns the photo image from the database as a JPEG file. This method is called by multiple views to display the image.

The main tasks for this exercise are as follows:

1. Create a photo controller.
2. Create the Index action.
3. Create the Display action.
4. Write the Create actions for GET and POST HTTP verbs.
5. Create the Delete actions for GET and POST HTTP verbs.
6. Create the GetImage action.

▶ Task 1: Create a photo controller.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the PhotoSharingApplication.sln file from the following location:
3. File path: **Allfiles (D):\Labfiles\Mod04\Starter\PhotoSharingApplication**
4. Create a new controller for handling **photo** objects by using the following information:
   - Controller name: **PhotoController**
   - Template: **Empty MVC controller**
5. Add **using** statements to the controller for the following namespaces:
   - **System.Collections.Generic**
   - **System.Globalization**
   - **PhotoSharingApplication.Models**
6. In the PhotoController class, create a new private object by using the following information:
   - Scope: **private**
   - Class: **PhotoSharingContext**
   - Name: **context**

   Instantiate the new object by calling the **PhotoSharingContext** constructor.

▶ Task 2: Create the Index action.

1. Edit the code in the **Index** action by using the following information:

- Return class: **View**
- View name: **Index**
- Model: **context.Photos.ToList()**

▶ Task 3: Create the Display action.

1. Add a method for the **Display** action by using the following information:
   - Scope: **public**
   - Return Type: **ActionResult**
   - Name: **Display**
   - Parameters: One integer called **id**
2. Within the **Display** action code block, add code to find a single **photo** object from its **ID**.
3. If no photo with the right ID is found, return the **HttpNotFound** value.
4. If a photo with the right ID is found, pass it to a view called **Display**.

▶ Task 4: Write the Create actions for GET and POST HTTP verbs.

1. Add a method for the **Create** action by using the following information:
   - Scope: **public**
   - Return type: **ActionResult**
   - Name: **Create**
2. Add code to the **Create** action that creates a new Photo and sets its **CreatedDate** property to today's date.
3. Pass the new **Photo** to a view called **Create**.
4. Add another method for the **Create** action by using the following information:
   - HTTP verb: **HTTP Post**
   - Scope: **public**
   - Return type: **ActionResult**
   - Name: **Create**
   - First parameter: a **Photo** object called **photo**.
   - Second parameter: an **HttpPostedFileBase** object called **image**.
5. Add code to the **Create** action that sets the **photo.CreatedDate** property to today's date.
6. If the **ModelState** is not valid, pass the **photo** object to the **Create** view. Else, if the image parameter is not null, set the **photo.ImageMimeType** property to the value of **image.ContentType**, set the **photo.PhotoFile** property to be a new byte array of length, **image.ContextLength**, and then save the file that the user posted to the **photo.PhotoFile** property by using the **image.InputStream.Read()** method.
7. Add the **photo** object to the context, save the changes, and then redirect to the **Index** action.

▶ Task 5: Create the Delete actions for GET and POST HTTP verbs.

1. Add a method for the **Delete** action by using the following information:
   - Scope: **public**

- Return type: **ActionResult**
- Name: **Delete**
- Parameter: an integer called **id**

2. In the **Delete** action, add code to find a single **photo** object from its **id**.
3. If no Photo with the right **id** is found, return the **HttpNotFound** value
4. If a Photo with the right **id** is found, pass it to a view called **Delete**.
5. Add a method called **DeleteConfirmed** by using the following information:
   - HTTP verb: **HTTP Post**
   - ActionName: **Delete**
   - Scope: **public**
   - Return type: **ActionResult**
   - Name: **DeleteConfirmed**
   - Parameter: an integer called **id**
6. Find the correct **photo** object from the **context** by using the **id** parameter.
7. Remove the **photo** object from the **context**, save your changes and redirect to the **Index** action.

▶ Task 6: Create the GetImage action.

1. Add a method for the **GetImage** action by using the following information:
   - Scope: **public**
   - Return type: **FileContentResult**
   - Name: **GetImage**
   - Parameter: an integer called **id**
2. Find the correct **photo** object from the **context** by using the **id** parameter.
3. If the **photo** object is not null, return a **File** result constructed from the **photo.PhotoFile** and **photo.ImageMimeType** properties, else return the **null** value.
4. Save the file.

Results: After completing this exercise, you will be able to create an MVC controller that implements common actions for the Photo model class in the Photo Sharing application.

Exercise 2: Optional—Writing the Action Filters in a Controller

Scenario

Your development team is new to MVC and is having difficulty in passing the right parameters to controllers and actions. You need to implement a component that displays the controller names, action names, parameter names, and values in the Visual Studio Output window to help with this problem. In this exercise, you will create an action filter for this purpose.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Add an action filter class.

2. Add a logValues method to the action filter class.

3. Add a handler for the OnActionExecuting event.

4. Register the Action Filter with the Photo Controller.

▶ Task 1: Add an action filter class.

1. Create a new class for the action filter by using the following information:
   - Name: **ValueReporter**
   - Folder: **Controllers**
2. Add **using** statements to the controller for the following namespaces:
   - **System.Diagnostics**
   - **System.Web.Mvc**
   - **System.Web.Routing**
3. Ensure that the **ValueReporter** class inherits from the **ActionFilterAttribute** class.

▶ Task 2: Add a logValues method to the action filter class.

1. Add a method to the **ValueReporter** class by using the following information:
   - Scope: **private**
   - Return type: **void**
   - Name: **logValues**
   - Parameter: a **RouteData** object called **routeData**.
2. Within the **logValues** method, call the **Debug.WriteLine** method to send the name of the controller and action to the Visual Studio Output window. For the category, use the string, "Action Values".
3. Within the **logValues** method, create a **foreach** loop that loops through the **var** items in **routeData.Values**.
4. In the **foreach** loop, call the **Debug.WriteLine** method to send the key name and value to Visual Studio Output window.

▶ Task 3: Add a handler for the OnActionExecuting event.

1. In the **ValueReporter** action filter, override the **OnActionExecuting** event handler.
2. In the **OnActionExecuting** event handler, call the **logValues** method, and pass the **filterContext.RouteData** object.
3. Save the file.

▶ Task 4: Register the Action Filter with the Photo Controller.

1. Open the **PhotoController** class and add the **ValueReporter** action filter to the **PhotoController** class.
2. Save the file.

Results: After completing this exercise, you will be able to create an action filter class that logs the details of actions, controllers, and parameters to the Visual Studio Output window, whenever an action is called.

Exercise 3: Using the Photo Controller

Scenario

In this exercise, you will:

- Create a temporary index and display views by using the scaffold code that is built into the Visual Studio MVC application template.
- Use the views to test controllers, actions, and action filters, and run the Photo Sharing application.

The main tasks for this exercise are as follows:

1. Create the Index and Display views.
2. Use the GetImage action in the Display view.
3. Run the application and display a photo.

▶ Task 1: Create the Index and Display views.

1. Compile the PhotoSharingApplication project to build the solution.
2. Add a new view to the **Index** action method of the **PhotoController** class by using the following information:
   - Folder: **Views/Photo**
   - Name: **Index**
   - View type: **Strong**
   - Model class: **Photo**
   - Scaffold template: **List**
3. Add a new view to the **Display** action method of the **PhotoController** class by using the following information:
   - Folder: **Views/Photo**
   - Name: **Display**
   - View type: **Strong**
   - Model class: **Photo**
   - Scaffold template: **Details**

▶ Task 2: Use the GetImage action in the Display view.

1. In the Display.cshtml code window, after the code that displays the **model.Title** property, add a code that runs if the **Model.PhotoFile** property is not **null**.
2. Within the **if** code block, render an <img> tag. Use the following information:
   - Tag: **<img>**
   - Width: **800px**
   - Source: **Blank**
3. In the **src** attribute of the <img> tag, add a call to the **Url.Action** helper by using the following information:
   - Controller: **Photo**
   - Action: **GetImage**

- Parameters: **Model.PhotoID**

4. Save the file.
5. Build the solution.

▶ Task 3: Run the application and display a photo.

1. Start debugging the application and access the following relative path:
   - Path: **/photo/index**
2. If you completed Exercise 2, in the Output pane of the **PhotoSharingApplication - Microsoft Visual Studio** window, locate the last entry in the **Action Values** category to verify whether there are any calls to the **Display** and the **GetImage** actions.
3. Display an image.
4. If you completed Exercise 2, in the Output pane of the **PhotoSharingApplication - Microsoft Visual Studio** window, locate the last entry in the **Action Values** category to verify whether there are any calls to the **Display** and the **GetImage** actions.
5. Stop debugging and close Microsoft Visual Studio.

Results: After completing this exercise, you will be able to create an MVC application with views that you can use to test controllers, actions, and action filters.

Question: What will happen if you click the Edit or Delete links in the Index view in the Lab?

Question: Why did you use the **ActionName** annotation for the **DeleteConfirmed** action in the **PhotoController** class?

Question: In the lab, you added two actions with the name, **Create**. Why is it possible to add these actions without using the ActionName annotation?

Module Review and Takeaways

In this module, you have seen the pivotal role that controllers and actions take in the construction of an MVC web application. Controllers and actions ensure that the application responds correctly to each user request, create instances of model classes, and pass the model class to a view for display. In the next module, you will see how to create and code views to implement the user interface for your application.

 **Best Practice:** Unless you have a good reason not to, keep to the convention that each controller should be named to match the corresponding model class, with "Controller" appended. For example, for the Photo model class, the controller should be called PhotoController. By maintaining this convention, you create a logically named set of model and controller classes, and can use the default controller factory.

 **Best Practice:** Take great care if you choose to override the built-in AuthorizeAttribute filter because it implements permissions and security for the web application. If you carelessly modify the security infrastructure of the MVC framework, you may introduce vulnerabilities in your application. Instead, use the built-in filter wherever possible.

Review Question(s)

Question: You want to ensure that the **CreatedDate** property of a new **Photo** object is set to today's date when the object is created. Should this value be set in the **Photo** model class constructor method or in the **PhotoController Create** action method?

Module 05

Developing ASP.NET MVC 4 Views

Contents:

Module Overview

Views are one of the three major components of the MVC programming model. You can construct the user interface for your web application by writing views. A view is a mixture of HTML markup and code that runs on the web server. You need to know how to write the HTML and code found in a view and use the various helper classes built into MVC. You also need to know how to write partial views, which render sections of HTML that can be re-used in many places in your web application. The knowledge of creating views will help you develop webpages that present dynamic content.

Objectives

After completing this module, you will be able to:

- Create an MVC view and add Razor markup to it to display data to users.
- Use HTML helpers in a view to render controls, links, and other HTML elements.
- Re-use Razor markup in multiple locations throughout an application.

Lesson 1
Creating Views with Razor Syntax

When a user makes a request to an MVC web application, a controller responds. Often, the controller action instantiates a model object, for example, a **Product**. The controller might get the **Product** by querying the database, by creating a new **Product**, or by performing some other step. To display the **Product**, the controller creates a view and passes the **Product** to it. The view builds a webpage by inserting properties from the **Product**, and sometimes, from other sources, into HTML markup. MVC sends the completed webpage to the browser. To create views, you must understand how MVC interprets the code you place in views and how a completed HTML page is built. By default, the Razor view engine performs this interpretation, but you can also use other view engines. If you take time to understand Razor, you will find it to be an excellent and versatile view engine that gives a great degree of control over the rendered HTML. Other view engines have very different syntaxes, which you may prefer if you have worked with earlier versions of ASP.NET or other server-side web technologies, such as Ruby on Rails.

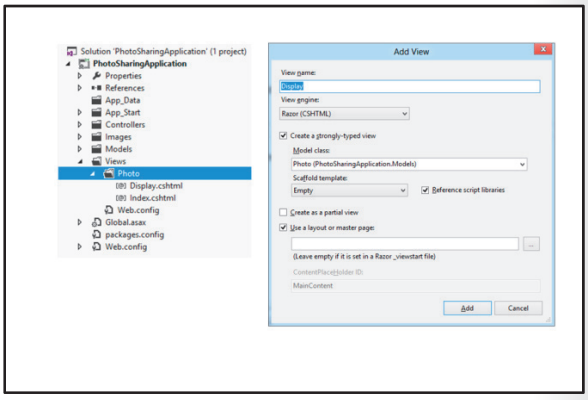
Lesson Objectives

After completing this lesson, you will be able to:

- Add views to an MVC 4 web application.
- Use the @ symbol in a Razor view to treat code as content.
- Describe the features of Razor syntax.
- Bind views to model classes to display data.
- Explain how to render HTML that is accessible to visitors with low vision.
- List a few alternative view engines to the Razor view engine.

Adding Views

In an MVC application, there is usually one controller for every model class. For example, a model class named **Product** usually has a controller called **ProductController** that contains all the actions relevant to products. There may be some controllers that do not correspond to any model classes, such as the **HomeController**. However, each controller can have multiple views. For example, you may want to create the following views for **Product** objects:



- *Details*. The details view can display a **Product**, its price, catalog number, and other details.
- *Create*. The create view can enable users to add a new **Product** in the catalog.
- *Edit*. The edit view can enable users to modify the properties of an existing **Product**.
- *Delete*. The delete view can enable users to remove a **Product** from the catalog.
- *Index*. The index view can display all the **Product** objects in the catalog.

By convention, an MVC web application creates all views within the top-level **Views** folder. Within this folder, there is a folder for each controller in the application. In the preceding example, the **Views** folder would contain a **Product** folder. This folder would contain the Details, Create, Edit, Delete, and Index views.

If you use the Razor view engine and the C# language, views are files with a .cshtml extension. If you use the Razor view engine and the Visual Basic language, views are files with a .vbhtml extension. Other view engines use different extensions.

How to Create a View File

You can add a view file in Visual Studio by performing the following steps:

1. In the **Solution Explorer** pane, select the folder where you want to create the view. For example, for a controller named **ProductController**, MVC expects view files to be located in the **Views/Product** folder.
2. Right-click the selected folder, point to **Add**, and then select **View**.

Alternatively, you can create a view file for a particular controller action by opening the controller code file, right-clicking the action, and then clicking **Add View**. If you create a view by using the controller code file, some properties in the **Add View** dialog box are filled in by default.

The following table describes the properties that you must complete in the **Add View** dialog box.

Property	Description
View name	This is the name of the view. The view file name is this name with the appropriate extension added. The name you choose should match the name returned by the corresponding controller action. If the action controller does not specify the name of the view to use, MVC assumes the name of the view matches the name of the controller action.
View engine	By default, you can select the Razor or ASPX view engines in this list. Other engines may be available if you have installed them.
Create a strongly-typed view	When you select this check box, you bind the view to a specific model class, such as a **Product** class. When you use strongly-typed views, you get additional IntelliSense and error checking help in Visual Studio as you write code in the view.
Model class	If you create a strongly-typed view, you need to specify the model class to bind to the view. Visual Studio will use this class when it formulates IntelliSense prompts and checks for compile-time errors.
Scaffold template	A scaffold template is a basic view that Visual Studio can use to create the view. If you specify a model class for the view, Visual Studio can create simple templates for Create, Edit, Details, Delete, and List views. When you become more experienced in using views, you may prefer to build a view from the empty scaffold template, rather than use populated scaffold templates.
Reference script libraries	When you select this check box, links to common client-side script files are included in the view. These links include the jQuery JavaScript library.
Create as a partial view	A partial view is a section of Razor code that you can re-use in multiple views in your application.
Use a layout or master page	A layout or master page can be used to impose a standard layout and

Property	Description
	branding on many pages within the web application.

Note: You will see other view engines later in this module.

Question: You are using the Razor view engine and Visual Basic to create views. You right-click the **Delete** action in the **CustomerController.vb** file. What is the name of the view file that Visual Studio will create by default?

Differentiating Server Side Code from HTML

The Razor view engine interprets view files and runs any server-side code contained in the view files. To do this, Razor must distinguish server-side code from HTML content that should be sent to the browser unchanged.

The Razor view engine looks for the @ symbol to identify server-side code.

In the following code example, Razor runs the line with the @ symbol as C# code on the web server. This code example displays the value of the **Title** property of the model class.

- Razor identifies server-side code by looking for the **@** symbol.
- In Razor syntax, the **@** symbol has various uses. You can:
  - Use **@** to identify server-side C# code.
  - Use **@@** to render an @ symbol in an HTML page.
  - Use **@:** to explicitly declare a line of text as content and not code.
  - Use **<text>** to explicitly declare several lines of text as content and not code.
- To render text without HTML encoding, you can use the **Html.Raw()** helper.

Using @ to Declare Server-Side Code

```
<p>Razor will not interpret any character in this line as code, because there is no “at”
symbol</p>
<p>
   The photo title is:
   @Model.Title
</p>
```

A section of code marked with the @ symbol is referred to as a Razor code expression. In Razor syntax, you mark the start of a Razor code expression with the @ symbol, but unlike the ASPX view engine, Razor lacks an expression ending symbol. Instead, Razor infers the end of a code expression by using a fairly sophisticated parsing engine.

Modifying the Interpretation of Code and Content

Sometimes, you may need to modify the logic that Razor uses to interpret code expressions. For example, if you want to display an @ symbol in the browser, you use @@. Razor interprets this as an escape sequence and renders a single @ symbol. This technique is useful for rendering email addresses.

Razor has sophisticated logic to distinguish code from content and often the @ symbol is all that is required. However, occasionally, you may find that an error occurs because Razor misinterprets content as code. To fix such errors, you can use the @: delimiter, which explicitly declares a line as content and not code.

If you want to declare several lines as content, use the <text> tag instead of the @: delimiter. Razor removes the <text> and </text> tags before returning the content to the browser.

In the following code example, Razor interprets the @: delimiter to display plain text.

Using the @: Delimiter

```
@(
@: This text is in a code block but the delimiter declares it as content.
)
```

HTML Encoding

For security reasons, when Razor runs any server-side code that returns a string, it encodes the string as HTML before rendering it to the browser. For example, you may have a **Comment** model class with the following properties:

- *Subject*. This property contains a subject for the comment as a simple string with no formatting information.
- *HtmlComment*. This property contains the comment text itself. More important, it includes the formatting information as HTML tags. For example, it might include <i> tags to italicize a word.

If you have used the normal **@Model.HtmlComment** syntax to return the formatted comment text, Razor encodes "<" as "&lt" and ">" as "&gt" in the HTML source. This results in the user seeing "<i>" on the webpage. This is not the behavior you intend because you want the users to see italicized text.

To render text without HTML encoding, use the **Html.Raw()** helper. For example, use **@Html.Raw(Model.HtmlComment)**.

Note: Razor encodes text as HTML for a specific security reason—to stop malicious users from injecting malicious code into your web application. If such attacks are successful, injected code may circumvent security restrictions in your web application and deny service to users, or it may access sensitive information. Therefore, you should only use the **Html.Raw()** helper when it is absolutely necessary. Also, if you must use the **Html.Raw()** helper, you must ensure that it does not operate on unchecked strings entered by users.

Features of Razor Syntax

Razor includes many useful features that you can use to control the manner in which ASP.NET MVC renders your view as HTML. These features include the following:

- A sample code block displaying the features of Razor.

```
@* Some more Razor examples *@

<span>
Price including Sale Tax: @Model.Price * 1.2
</span>
<span>
Price including Sale Tax: @(Model.Price * 1.2)
</span>

@if (Model.Count > 5)
{
 <ol>
  @foreach(var item in Model)
  {
    <li>@item.Name</li>
  }
 </ol>
}
```

Razor Comments

You may want to include comments in your Razor code to describe it to other developers in your team. This is an important technique that improves developer productivity by making it easier to understand code.

You can declare a Razor comment by using the **@*** delimiter, as shown in the following code example.

A Razor Comment

```
@* This text will not be rendered by the Razor view engine, because this is a comment. *@
```

Implicit Code Expressions and Parentheses

Razor uses implicit code expressions to determine parts of a line that are server-side code. Usually, implicit code expressions render the HTML you want, but occasionally, you might find that Razor interprets an expression as HTML when it should be run as server-side code.

For example, if the **Product.Price** property contains the value 2.00, the following code renders "Price Including Sales Tax: 2.00 * 1.2".

An Implicit Code Expression

```
<span>Price Including Sales Tax: @Model.Price * 1.2</span>
```

You can control and alter this behavior by using parentheses to enclose the expression for Razor to evaluate.

For example, if the **Product.Price** property contains the value 2.00, the following code renders "Price Including Sales Tax: 2.40".

Using Parentheses to Explicitly Delimit Expressions

```
<span>Price Including Sales Tax: @(Model.Price * 1.2)</span>
```

Razor Code Blocks, Loops, and Conditions

If you want to write multiple lines of server-side code, you can do it without prefacing every line with the @ symbol by using a Razor code block.

A Razor Code Block

```
@ {
   //Razor interprets all text within these curly braces as server-side code.
}
```

Razor also includes code blocks that run conditional statements or loop through collections.

For example, Razor runs the following code only if the **Product.InStock** Boolean property returns **true**.

A Razor If Code Block

```
@if (Model.InStock)
{
   Html.ActionLink("Buy This Product", "AddToCart")
}
```

Razor loops are useful for creating index views, which display many objects of a particular model class. A product catalog page, which displays many products from the database, is a good example of an index view. To implement an index view, a controller action passes an enumerable collection of objects to the view.

You can loop through all the objects in an enumerable collection by using the **foreach** code block:

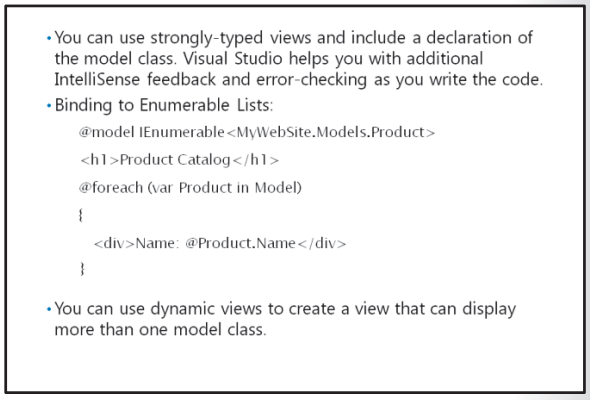
A Razor Foreach Code Block

```
@foreach (var item in Model)
{
   <div>@item.ProductName</div>
}
```

Question: You want to describe a code block to developers in your view file. You do not want your description to be passed to the browser. What syntax should you use?

Binding Views to Model Classes and Displaying Properties

Many Razor views are designed to display properties from a specific model class. If you bind such views to the class they display, you get extra help such as IntelliSense feedback as you write the Razor code. Other views may display properties from different model classes in different cases, or may not use a model class at all. These views cannot be bound to a model class and are called dynamic views. You must understand how to write Razor code for both these situations.



Strongly-Typed Views

In many cases, when you create a view, you know that the controller action will always pass an object of a specific model class. For example, if you are writing the **Display** view for the **Display** action in the **Product** controller, you know from the action code that the action will always pass a **Product** object to the view.

In such cases, you can create a strongly-typed view. A strongly-typed view is a view that includes a declaration of the model class. When you declare the model class in the view, Visual Studio helps you with additional IntelliSense feedback and error-checking as you write the code. This is possible because Visual Studio can check the properties that the model class includes. Troubleshooting run-time errors is also easy. You should create strongly-typed views whenever you can so that you benefit from this extra IntelliSense and error-checking. If you follow this as a best practice, you will make fewer coding errors. A strongly-typed view only works with an object of the model class in the declaration.

To create a strongly-typed view in Visual Studio, select **Create a strongly typed view** in the **Add View** dialog box, and then choose the model class to bind to the view.

You can choose the **Product** model class so that Visual Studio can bind the view to the class you specify, by adding the following line of code at the top of the view file.

The Model Declaration

```
@model MyWebSite.Models.Product
```

Later in the view file, you can access properties of the model object by using the **Model** helper. For example, you might access a product catalog number in this manner: **@Model.CatalogNumber**.

Binding to an Enumerable List

Some views display many instances of a model class. For example, a product catalog page displays many instances of the **Product** model class. In such cases, the controller action passes a list of model objects to the view, instead of a single model object. You can still create a strongly-typed view in this situation, but you must use a different model declaration. You usually loop through the items in the list by using a Razor **foreach** loop.

The following code example shows a multiple-item view with the model declaration and the loop that enumerates all the **Product** objects passed by the action.

A View of a List of Objects

```
@model IEnumerable<MyWebSite.Models.Product>
<h1>Product Catalog</h1>
@foreach (var Product in Model)
{
   <div>Name: @Product.Name</div>
```

```
}
```

Using Dynamic Views

Sometimes, you might want to create a view that can display more than one model class. For example, you might want to create a view that can display your own products and products from a different supplier, together. These model classes might be named **Product** and **ThirdPartyProduct**. Some properties can be similar while others differ. For example, both **Product** and **ThirdPartyProduct** might include the **Price** property while only the **ThirdPartyProduct** model class includes the **Supplier** property.

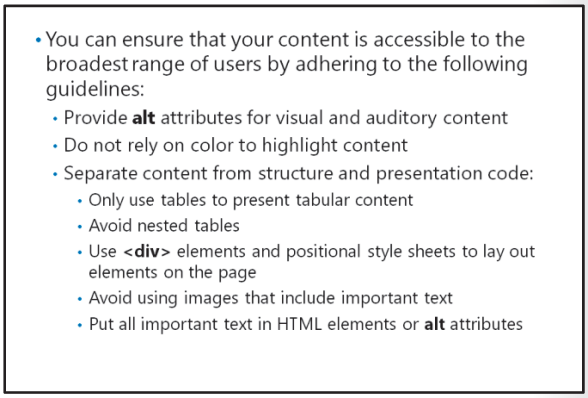
Furthermore, sometimes, the controller action does not pass any model class to the view. The most common example of such a classless view is the site home page.

In such cases, you can create a dynamic view. A dynamic view does not include the **@model** declaration at the top of the page. You can choose to add the **@model** declaration to change the view into a strongly-typed view, later.

When you create dynamic views, you receive less helpful IntelliSense feedback and error checking because Visual Studio cannot check the model class properties to verify the code. In such scenarios, you have to ensure that you access only the properties that exist. To access a property that may or may not exist, such as the **ThirdPartyProduct.Supplier** property in the preceding example, check the property for **null** before you use it.

Question: You want to write a view that displays ten objects of the **Photo** model class. What model declaration should you use?

Rendering Accessible HTML



The Internet is for everyone, regardless of any disabilities an individual user may have. Furthermore, if users with disabilities cannot easily browse your website, they may visit your competitors' websites and your company may lose business. You should therefore ensure that people with disabilities—such as those who have low vision or are hard-of-hearing—can use your web application, and that their web browsers can parse the HTML content that your site presents. Because MVC views are responsible for rendering HTML in an MVC web application, you render accessible HTML by writing views that follow accessibility guidelines. When you write views, keep the following challenges and best practices in mind.

Users have different requirements depending on their abilities and disabilities. For example, consider the following factors:

- Users with low vision may use a standard browser, but they may increase text size with a screen magnifier so that they can read the content.
- Profoundly blind users may use a browser with text-to-speech software or text-to-Braille hardware.
- Color-blind users may have difficulty if a color difference is used to highlight text.
- Deaf users may not be able to access audio content.

- Users with limited dexterity may find it difficult to click small targets.
- User with epilepsy may have seizures if presented with flashing content.

You can ensure that your content is accessible to the broadest range of users by adhering to the following guidelines:

- Do not rely on color differences to highlight text or other content. For example, links should be underlined or formatted in bold font to emphasize them to color-blind users.
- Always provide equivalent alternative content for visual and auditory content. For example, always complete the **alt** attribute for images. Use this attribute to describe the image so that text-to-speech software or text-to-Braille hardware can render meaningful words to the user.
- Use markup and style sheets to separate content from structure and presentation code. This helps text interpretation software to render content to users without being confused by structural and presentation code. For example, you should apply the following best practices to display content on your webpage:
  - Avoid using tables to display the content. You should use tables only to present tabulated content. Tables can be used to render graphics and branding on a webpage, but in an accessible site, use positional style sheets to display content. Text readers do not read style sheets.
  - Avoid using nested tables. In a nested table, a table cell contains another table. These are particularly confusing to text readers because they read each table cell in a sequential order. The user is likely to become disoriented and unable to determine which cell is being read and what it means.
  - Avoid using images that include important text. Text readers cannot render text from within an image file. Instead, use markup to render this text.

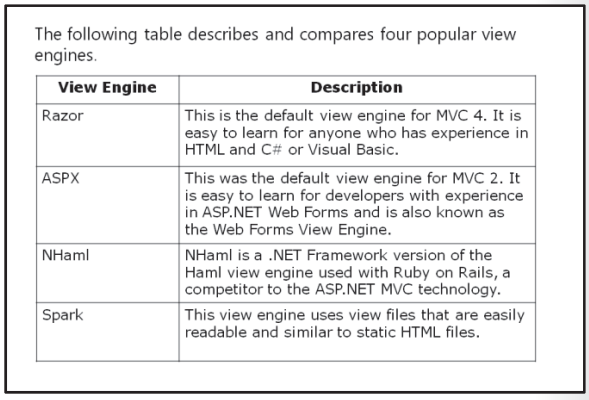
Additional Reading: The World Wide Web Consortium (W3C) has a project called the Web Accessibility Initiative (WAI) that promotes accessible web content. This project has published the Web Content Accessibility Guidelines (WCAG). These guidelines are accepted by the web publishing industry as definitive. To read the full guidelines, go to: http://go.microsoft.com/fwlink/?LinkID=288955&clcid=0x409

Note: In ASP.NET MVC, the developer has complete control over the HTML that the web server sends to the web browser. However, you must understand accessibility principles to write accessible MVC views. Ensure that your entire development team is familiar with the requirements of accessibility and the related best practices.

Question: You want to present your company logo at the uppermost part of the page. The logo is a .gif file that includes the name of the company in an unusual font. How can you ensure that the logo is accessible?

Alternative View Engines

A view engine is a component of the MVC framework that is responsible for locating view files, running the server-side code they contain, and rendering HTML that the browser can display to the user. Many views work with the Razor view engine. Razor is the default view engine in MVC 4 and is a highly flexible and efficient view engine.


The following table describes and compares four popular view engines.

View Engine	Description
Razor	This is the default view engine for MVC 4. It is easy to learn for anyone who has experience in HTML and C# or Visual Basic.
ASPX	This was the default view engine for MVC 2. It is easy to learn for developers with experience in ASP.NET Web Forms and is also known as the Web Forms View Engine.
NHaml	NHaml is a .NET Framework version of the Haml view engine used with Ruby on Rails, a competitor to the ASP.NET MVC technology.
Spark	This view engine uses view files that are easily readable and similar to static HTML files.

However, some developers may prefer to use a different view engine for the following reasons:

- *View Location Logic*. Razor assumes that all views are located in the **Views** top-level folder. Within this, Razor assumes that each controller has a separate folder. For example, views for the **ProductController** are located in the **Product** folder within **Views**. Razor also looks for some views in the **Shared** subfolder. These views can be used by more than one controller. If you do not like this view location logic, you can use an alternate view engine, or create a custom view engine.
- *View Syntax*. Each view engine uses a different syntax in view files. For example, in Razor, the **@** symbol delimits server-side code. By contrast, in the ASPX view engine, **<% %>** delimiters are used. This is familiar to developers who have worked with Web Forms. You may prefer one syntax over another for the following reasons:
  - *Experience*. The technologies that you or your team have previously worked with may make it easier for you to learn the syntax of a specific view engine.
  - *Readability*. Some view engines work with view files that are easily readable for anyone who knows HTML. Web designers, who do not know C# or other programming languages, may be able to learn readable syntaxes.
  - *Succinctness*. Some view engines work with view files that are terse and generate HTML with relatively small amounts of code.

Common View Engines

The following table describes and compares four popular view engines.

View Engine	Description	Code Example

Razor	This is the default view engine for MVC 4. It is easy to learn for anyone who has experience in HTML and C# or Visual Basic. Visual Studio provides good IntelliSense feedback for Razor code and it is compatible with unit testing.	`@model IEnumerable<MyWebSite.Models.Product>`
		
		`<ul>`
		`@foreach (var p in Model)`
		`{`
		`  <li>Name: @p.Name</li>`
		`}`

ASPX	This was the default view engine for MVC 1 and 2. It is easy to learn for developers with experience in ASP.NET Web Forms and it is also known as the Web Forms View Engine. It includes good IntelliSense feedback.	`<%@ Control Inherits=`
		`"System.Web.Mvc.ViewPage<IEnumerable<Product>>"`
		`%>`
		`<ul>`
		`<% foreach(var p in model){%>`
		`  <li><%=p.Name%></li>`
		`<%}%>`

View Engine	Description	Code Example
		`</ul>`

NHaml	NHaml is a .NET Framework version of the Haml view engine used with Ruby on Rails, a competitor to ASP.NET MVC. Developers who have developed websites with Ruby on Rails may prefer this view engine. Visual Studio does not provide IntelliSense feedback for this view engine. NHaml syntax is terse and efficient, but often less intelligible to inexperienced developers.	`@type=IEnumerable<Product>`
		`%ul`
		`  - foreach (var p in model)`
		`    %li= p.Name`

Spark	This view engine uses view files that are easily readable and similar to static HTML files. This design results in view files that are readable for anyone who knows HTML. You can obtain IntelliSense support for Spark in Visual Studio.	`<viewdata products="IEnumerable[[Product]]"/>`
		`<ul if="products.Any()">`
		`    <li each="var p in products">${p.Name}</li>`
		`</ul>`

Additional Reading: For more information about alternative view engines, go to:
http://go.microsoft.com/fwlink/?LinkID=288956&clcid=0x409
http://go.microsoft.com/fwlink/?LinkID=288957&clcid=0x409

Creating Custom View Engines

You can also create your own view engine to implement custom logic for locating views or to implement custom syntax for view files. Custom view engines are rarely created because the existing view engines are powerful, flexible, and easy to learn. The default logic for finding view files is also simple and rarely needs modification. To create a custom view engine, you must take the following steps:

- *Create a View Class*. This class must implement the **IView** interface and have a **Render()** method. You need to write the custom logic in the **Render()** method for interpreting custom syntax. This code is likely to include regular expressions and other sophisticated string-parsing code.
- *Create a View Engine*. This class should inherit the **VirtualPathProviderViewEngine** class and include the **CreateView()** and **CreatePartialView()** methods. By inheriting from the **ViewPathProviderViewEngine** class, you can modify the **ViewLocationFormat** property to implement custom logic for locating views.
- *Register the Custom View Engine*. To register your custom view engine with MVC, call **ViewEngines.Engines.Add()** in the Global.asax file.

 **Additional Reading:** For more information about creating custom view engines in ASP.NET MVC page, see: http://go.microsoft.com/fwlink/?LinkID=288958&clcid=0x409

Question: Is there any HTML markup that the Razor view engine cannot render?

Lesson 2
Using HTML Helpers

The ASP.NET MVC 4 Framework includes many HTML helper functions that you can use in views. HTML helpers are C# methods that you can call to render values, labels, and input controls such as text boxes. Helpers include common logic that makes it easier for you to render HTML for MVC model class properties and other tasks. Some HTML helpers examine the definition of a property in a model class and then render appropriate HTML. Some HTML helpers call controller actions and others validate user input data. You can also create custom HTML helpers. You need to know the appropriate HTML helper to use in any scenario and how to pass the correct parameters to relevant HTML helpers.

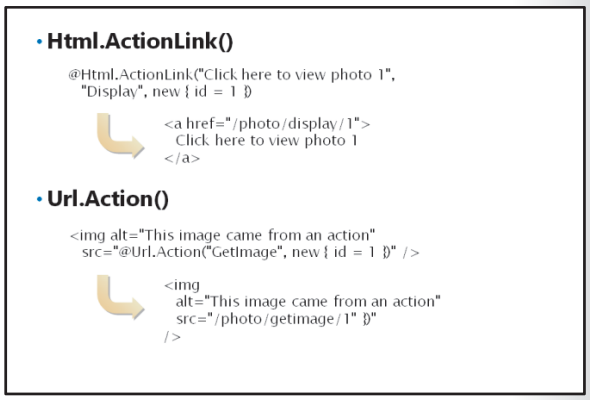
Lesson Objectives

After completing this lesson, you will be able to :

- Use the **Html.ActionLink** and **Url.Action** helpers to call a controller action.
- Use the **Html.DisplayNameFor** and **HtmlDisplayFor** helpers to display the value of a model class property.
- Use the **Html.LabelFor** and **HTML.EditorFor** helpers to render an editor control for a model class property.
- Use the **Html.BeginForm** helper to render a form for a controller action.
- Use HTML helpers to validate data typed by users.
- Use various HTML helpers to build a user interface for a web application.

Using Action Helpers

HTML helpers are simple methods that in most cases return a string. This string is a small section of HTML that the view engine can insert into the view file to generate the completed webpage. You can write views that render any type of HTML content without using a single helper if you prefer not to use HTML helpers. However, HTML helpers make the task of managing HTML content easier by rendering common HTML strings for often-used scenarios.



The **Html.ActionLink()** and **Url.Action()** helpers enable you to render HTML that calls controller actions.

The Html.ActionLink() Helper

When the user makes a request to an MVC web application, the ASP.NET MVC 4 framework forwards the request to the correct controller and action, thereby passing the correct parameters. It is possible for users to make such requests by typing a URL into the Address bar of a web browser, but they rarely know the name of the controller, action, or parameters. Instead, they usually click a link. The **Html.ActionLink()** helper can be used to render links to actions for you. The helper returns an **<a>** element with the correct **href** parameter value when users click a link in a webpage.

In the following code example, the **Html.Actionlink()** helper renders an **<a>** element that calls the **Display** action in the **Photo** controller, and passes an integer for the photo ID.

Using the Html.ActionLink() Helper

```
@Html.ActionLink("Click here to view photo 1", "Display", new { id = 1 })
```

By default, the example renders the following HTML:

```
<a href="/photo/display/1">Click here to view photo 1</a>
```

Notice that you pass parameters to the helper by using an anonymous object. In this manner, you can pass multiple parameters.

Note: The **Html.ActionLink()** helper works with the routing engine to formulate the correct URL for the link. The helper will render the **href** value above, assuming it is called from a PhotoController view. There are several overloads of the ActionLink helper, however, if you wish to use it to call actions in other controllers.

The Url.Action Helper

The **Url.Action()** helper renders a URL by using the routing engine without rendering the enclosing **<a>** element.

You can use the **Url.Action()** helper to populate the **src** attribute of an **<img>** element or **<script>** element. This helper is useful whenever you want to formulate a URL to a controller action without rendering a hyperlink.

The following code example shows how to render an image by calling the **Url.Action** helper within the **src** attribute of an **img** element:

Rendering an Image from an Action

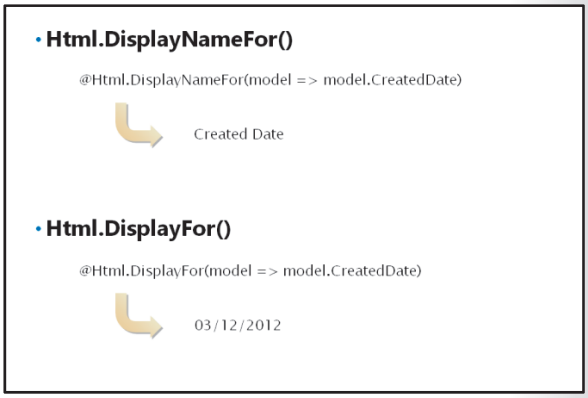
```
<img alt="This image came from an action" src="@Url.Action("GetImage", new { id = 1 })"
/>
```

Additional Reading: For more information about the **Html.ActionLink()** and **Url.Action()** helpers, see:
http://go.microsoft.com/fwlink/?LinkID=288959&clcid=0x409
http://go.microsoft.com/fwlink/?LinkID=288960&clcid=0x409

Question: You want to render an HTML5 **<audio>** tag to play a sound file from an action. Would you use the **Html.ActionLink()** helper or the **Url.Action()** helper?

Using Display Helpers

MVC includes several helpers that display properties from model classes. You can use these helpers to build views that display product details, comment details, user details, and so on. **Html.DisplayNameFor()** renders the name of a model class property. **Html.DisplayFor()** renders the value of a model class property. Both these helpers examine the definition of the property in the model class, including the data display annotations, to ensure that they render the most appropriate HTML.



The Html.DisplayNameFor() Helper

You use the **Html.DisplayNameFor()** helper to render the display name for a property from the model class. If your view is strongly-typed, Visual Studio checks whether the model class contains a property with the right name as you type the code. Otherwise, you must ensure that you use a property that exists or verify that it is not **null** before you use it. You specify the property of the model class to the **Html.DisplayNameFor()** HTML helper by using a lambda expression.

The following code example renders the display name of the **CreatedDate** property by using the **Html.DisplayNameFor()** HTML helper .

Using the Html.DisplayNameFor() Helper

```
@Html.DisplayNameFor(model => model.CreatedDate)
```

The text rendered by the **Html.DisplayNameFor()** helper depends on the model class. If you used a DisplayName annotation to give a more descriptive name to a property, then **Html.DisplayNameFor()** will use that DisplayName annotation value; otherwise, it will render the name of the property.

The Html.DisplayFor() Helper

The **Html.DisplayFor()** helper considers any display annotations that you specify in the model class, and then renders the value of the property.

In the following code example, the **Html.DisplayFor()** helper is used to render a date string.

Using the Html.DisplayFor() Helper

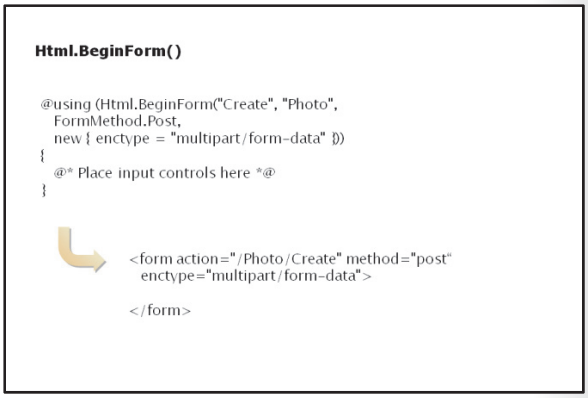
```
This was created on: @Html.DisplayFor(model => model.CreatedDate)
```

If you used the **DisplayFormat** annotation in the model class with DataFormatString set to "{0:dd/MM/yy}", then the **Html.DisplayFor()** helper ensures that the date is displayed in short date format.

Question: You want to ensure that a view displays "This product was last changed on" before the **ModifiedDate** property. This text is declared in the class with the **DisplayName** annotation. What code would you write in the view?

The Begin Form Helper

To accept user input, you must provide a form on your webpage. A typical form consists of a set of labels and input controls. The labels indicate to the user the property for which they must provide a value. The input controls enable the user to enter a value. Input controls can be text boxes, check boxes, radio buttons, file selectors, drop-down lists, or other types of control. There is usually a submit button and cancel button on forms.



Rendering HTML Forms

To build a form in HTML, you must start with a **<form>** element in the HTML webpage. All labels and input controls must be within the **<form>** element. In an MVC view, you can use the **Html.BeginForm()** helper to render this element and set the controller action to which the form sends data.

You can also specify the HTTP method that the form uses to submit data. If the form uses the POST method, which is the default, the browser sends form values to the web server in the body of the form. If the form uses the GET method, the browser sends form values to the web server in the query string in the URL.

In the rendered HTML, the **<form>** element must be closed with a **</form>** tag. In Razor views, you can ensure that the form element is closed with an **@using** code block. Razor renders the **</form>** tag at the closing bracket of the code block.

The following code example shows how to render a form that sends data to the **Register** action in the **CustomerController**. The form uses the POST HTTP verb.

Using the Html.BeginForm() Helper

```
@using (Html.BeginForm("Register", "Customer", FormMethod.Post))
{
   @* Place input controls here *@
}
```

Using Forms to Post Files

In HTML, if you want to enable website users to upload files, you should use an HTML form with special properties. Specifically, the **<form>** element should include the **enctype** attribute set to the value **multipart/form-data** and the form must use the POST HTTP verb.

You can specify additional attributes for the **<form>** element by passing them in an **htmlAttributes** parameter as new objects. This is a good method to use to add the **enctype** attribute. If the form includes any uploaded files, you must set the **enctype** attribute to **multipart/form-data**.

The following code example shows how to render a file upload form that sends data to the **Create** action in the **ProductController**. The form uses the POST verb and sets the **enctype** attribute.

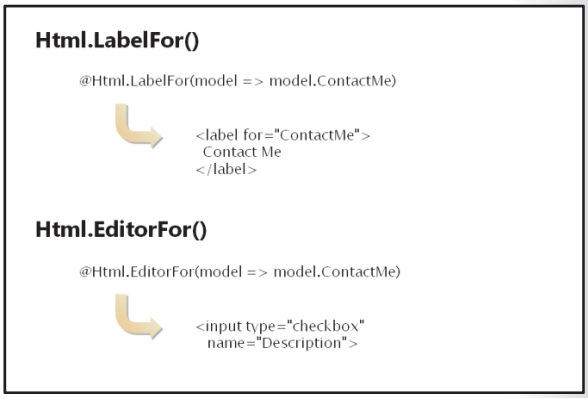
Using the Html.BeginForm() Helper

```
@using (Html.BeginForm("Create", "Photo", FormMethod.Post, new { enctype =
"multipart/form-data" }))
{
   @* Place input controls here *@
}
```

Question: You have created a form with a file selector control that uses the GET method. You have set the **enctype** attribute to **multipart/form-data**, but when you try to access the file in the action method, an exception is thrown. What have you possibly done incorrectly?

Using Editor Helpers

Within HTML forms, there are many HTML input controls that you can use to gather data from users. In Razor views, the **Html.LabelFor()** and **Html.EditorFor()** helpers make it easy to render the most appropriate input controls for the properties of a model class. To understand how these helpers render HTML, you must first understand the HTML input controls. Some common HTML controls are described in the following table.



Control	Example	Description
Text box	`<input type="text" name="Title">`	A single-line text box in which the user can enter a string. The **name** attribute is used to identify the entered value in the query string or to send form data by using the POST method.
Multi-line text box	`<textarea name="Description" rows="20" cols="80">`	A multi-line text box in which the user can enter longer strings.
Check box	`<input type="checkbox" name="ContactMe"> Please send me`	A check box submits a Boolean value.

Control	Example	Description
	`new offer details`	

 **Additional Reading:** For more information about all HTML form elements, go to: http://go.microsoft.com/fwlink/?LinkID=288961&clcid=0x409

The Html.LabelFor() Helper

The **Html.LabelFor()** helper is similar to the **Html.DisplayNameFor()** helper because it renders the name of the property that you specify, taking into account the **DisplayName** annotation if it is specified in the model class. However, unlike the **Html.DisplayNameFor()** helper, the **Html.LabelFor()** helper renders a **<label>** element.

The following code example renders a label for the **CreatedDate** property.

Using the Html.LabelFor() Helper

```
@Html.LabelFor(model => model.CreatedDate)
```

The Html.EditorFor() Helper

The **Html.EditorFor()** helper renders HTML **<input>** elements and other form controls for properties from an model class. This helper renders the most appropriate element for each property data type. For example, the **Html.EditorFor()** helper renders **<input type="text">** for a string property. If the string property is annotated with **[DataType(DataType.MultilineText)]**, the **Html.EditorFor()** helper renders a **<textarea>** element instead. The following table describes the HTML that **Html.EditorFor()** renders for different model classes.

Control	Model Class Property	HTML Rendered by EditorFor()
Text box	`public string Title { get; set; }`	`<input type="text" name="Title">`
Multi-line text box	`[DataType(DataType.MultilineText)]` `public string Description { get; set;}`	`<textarea name="Description"` `rows="20" cols="80">`

Check box	`[Display(Name = "Please send me new offer details")]`
	`public bool ContactMe { get; set; }`
		`<input type="checkbox"`
		`name="ContactMe">`
		`Please send me new offer details.`

If the action passes an existing model object to the view, the **Html.EditorFor()** helper also populates each control with the current values of each property.

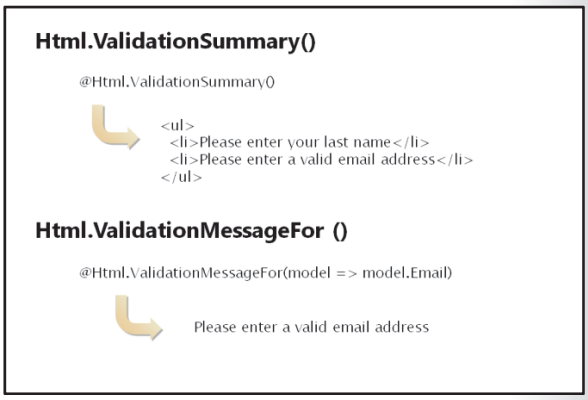
The following code example renders a check box for the **ContactMe** property.

Using the Html.EditorFor() Helper

```
@Html.EditorFor(model => model.ContactMe)
```

Question: You have a property in the **Product** model class named **ProductID**. You want to include this in the HTML page so that client-side script can use the **ProductID** value. However, you do not want the value to be displayed to users. In the model class, you have annotated the **ProductID** property with the **[HiddenInput(DisplayValue=false)]** attribute. How will the **Html.EditorFor()** helper render this property?

Using Validation Helpers



When you request information from users, you often want the data they enter in a certain format for further use in the web application. For example, you might want to ensure that users enter a value for the **LastName** property. You might also want to ensure that users enter a valid email address for the **EmailAddress** property. You can set these requirements by using validation data annotations in the model class. In controller actions, you can check the **ModelState.IsValid** property to verify if a user has entered valid data.

When users submit a form, if they have entered invalid data, most websites display validation messages. These messages are often highlighted in red, but other colors or formats can be used. Often, there is a validation summary at the top of the page such as "Please enter valid information for the following fields: Last Name, Email Address". The form may display detailed validation messages next to each control in which the user entered invalid data, or highlight the problems in user supplied data with an asterisk.

When your MVC view displays a form, you can easily show validation messages by using the validation helpers. Validation helpers render HTML only when users have entered invalid data.

The Html.ValidationSummary() Helper

Use the **Html.ValidationSummary()** helper to render a summary of all the invalid data in the form. This helper is usually called at the top of the form. When the user submits invalid data, the validation messages are displayed for each invalid field in a bulleted list. You can control the validation messages in the model class by using this syntax in the annotation:

```
[Required(ErrorMessage="Please enter your last name")]
```

The following code example renders a summary of validation messages.

Using the Html.ValidationSummary() Helper

```
@Html. ValidationSummary()
```

The Html.ValidationMessageFor() Helper

Use the **Html.ValidationMessageFor()** helper to render validation messages next to each input in the form.

The following code example renders a validation message for the **LastName** property of the model class.

Using the Html.ValidationMessageFor() Helper

```
@Html.ValidationMessageFor(model => model.LastName)
```

Demonstration: How to Use HTML Helpers

In this demonstration, you will see how to create a view and use HTML helpers to render controls in it. You will also see how validation requirements, which were set with data annotations in the model class, take effect in a view.

Demonstration Steps

1. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, expand **Controllers**, and then click **OperaController.cs**.
2. On the **BUILD** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Build Solution**.
3. In the **OperaController.cs** code window, locate the following code, right-click the code, and then click **Add View**.

```
public ActionResult Create ()
```

4. In the **View Name** box of the **Add View** dialog box, ensure that the name displayed is **Create**.
5. In the **Add View** dialog box, ensure that the **Create a strongly-typed view** check box is selected.
6. In the **Model class** box, ensure that the value is **Opera (OperasWebSite.Models)**. If not, in the **Model class** box, click **Opera (OperasWebSite.Models)**.
7. In the **Scaffold template** box, ensure that the value is **Empty**.
8. In the **Add View** dialog box, ensure that the **Use a layout or master page** check box is not selected, and then click **Add**.
9. In the **DIV** element of the **Create.cshtml** code window, type the following code.

```
<h2>Add an Opera</h2>
```

10. Place the mouse cursor at the end of the **</h2>** tag, press Enter twice, and then type the following code.

```
@using (Html.BeginForm(
   "Create", "Opera",
   FormMethod.Post))
{
}
```

11. In the **using** code block, type the following code.

```
<p>
@Html.LabelFor(model =>
      model.Title):
@Html.EditorFor(model =>
      model.Title)
   @Html.ValidationMessageFor(
      model => model.Title)
</p>
```

12. Place the mouse cursor at the end of the **</p>** tag corresponding to the **model.Title** property, press Enter twice, and then type the following code.

```
<p>
@Html.LabelFor(model =>
      model.Year):
@Html.EditorFor(model =>
      model.Year)
   @Html.ValidationMessageFor(
      model => model.Year)
</p>
```

13. Place the mouse cursor at the end of the **</p>** tag corresponding to the **model.Year** property, press Enter twice, and then type the following code.

```
<p>
   @Html.LabelFor(model =>
      model.Composer):
   @Html.EditorFor(model =>
      model.Composer)
   @Html.ValidationMessageFor(
      model => model.Composer)
</p>
```

14. Place the mouse cursor at the end of the **</p>** tag corresponding to the **model.Composer** property, press Enter twice, and then type the following code.

```
<input type="submit"
   value="Create" />
```

15. Place the mouse cursor at the end of the **<input>** tag, press Enter, and then type the following code.

```
@Html.ActionLink("Back to List", "Index")
```

16. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.

Note: The Operas I Have Seen page is displayed.

17. On the Operas I Have Seen page, click **operas I've seen**.

Note: On the Index page, the list of Operas is displayed.

18. On the Index page, click **Create New**.

Note: The Add an Opera page is displayed.

19. In the **Year** box of the Add an Opera page, type **1597**, and then click **Create**.

Note: Messages corresponding to the **Title**, **Year**, and **Composer** boxes are displayed. The web application mandates you to enter values in all the boxes. Alerts are also displayed for any inappropriate entries, with relevant messages.

20. In the **Title** box of the Add an Opera page, type **Rigoletto**.
21. In the **Year** box of the Add an Opera page, type **1851**.

22. In the **Composer** box of the Add an Opera page, type **Verdi**, and then click **Create**.

Note: The Opera is created with the mentioned values.

23. In the Windows Internet Explorer window, click the **Close** button.
24. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Lesson 3
Re-using Code in Views

In a web application, you frequently render similar blocks of HTML content in different webpages. For example, in an e-commerce web application, you might want to display the most popular products on the home page, on the front page of the product catalog, at the top of the product search page, and perhaps in other locations. To render such HTML content in an MVC application, you may copy and paste code from one Razor view into others. However, to change the display of top products later in the project, you would then have to make identical changes in many locations. A better practice is to use a partial view. A partial view renders a section of HTML content that can be inserted into several other views at run time. Because the partial view is a single file that is re-used in several locations, when you implement a change in a single location, the changes are updated in other locations in the web application. Partial views increase the manageability of MVC web applications and facilitate a consistent presentation of content throughout a web application.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how partial views enable you to re-use Razor code.
- Create partial views in an MVC 4 web application.
- Use partial views in an MVC 4 web application.
- Determine when to use partial views.

Creating Partial Views

You can use partial views to render identical HTML or similar HTML in different locations in your web application. Consider a web application in which users can comment on articles. In the article view, you want to display the comments for that specific article at the end of the page. You want a similar display on the home page, but you want to display the most popular comments on any article. You can build a single comments partial view for this scenario as follows:

You can use partial views to render the same HTML content in different locations in your web application

- Creating and Naming Partial Views:
  - Create a partial view by using the **Add View** dialog
  - Name partial views with an underscore prefix to keep to convention
- Strongly-typed and dynamic partial views:
  - Create strongly-typed partial views if you are certain that the partial view will always display the same model class
  - Create dynamic partial views if you are not sure if the partial view will always display the same model class

- First, create a partial view that displays a collection of comments in the required format.
- Next, create a controller action that retrieves the most popular comments, and then passes them to the partial view.
- Then, create another controller action that uses the article ID to find comments specific to that article. The controller action then passes this collection of comments to the partial view.
- Finally, call the appropriate actions from the home page and the article view.

Because the partial view renders a collection of comments, you can re-use it for various situations by passing different collections to it from controller actions.

Creating and Naming Partial Views

In Visual Studio, you can create a partial view by using the **Add View** dialog, in the same manner as creating any other view. Ensure that you select the **Create as a Partial View** check box. Because partial views render a small section of HTML rather than a complete webpage, Visual Studio does not add **<head>**, **<body>**, and other tags to the partial view. By convention, partial views are named prefixed with an underscore character, for example, **_CommentsList.cshtml**. This convention is optional but is often helpful to distinguish partial views in Solution Explorer.

Partial views are often created inside the **/Views/Shared** folder in your site. Views in the **Shared** folder are accessible to many controllers, whereas views in the **/Views/Comment** folder are accessible only from the **CommentController**.

Strongly-typed and dynamic partial views

Similar to other views, you can create strongly-typed partial views if you are certain that the partial view will always display the same model class. Visual Studio can provide the most informative IntelliSense feedback and error-checking for strongly-typed partial views, when compared with other types of views. A strongly-typed view has the declaration of the **@model** directive at the top of the file. Alternatively, you can create a dynamic partial view if you are not certain that the partial view will always display the same model class. You can create dynamic partial views by omitting the **@model** directive declaration.

In the following example of a partial view, you can see the **@model** declaration that makes the partial view strongly-typed. You can also see that the partial view renders only an unordered list and not a complete HTML webpage as the following lines of code indicate.

A Strongly-Typed Partial View

```
@model IEnumerable<Adworks.Models.Comment>
<ul>
   @foreach (var item in Model) {
      <li>
         Subject: @Html.DisplayFor(modelItem => item.Subject)
      </li>
   }
</ul>
```

Question: You want to create a partial view that displays a list of comments. The comments partial view will be called by actions in the **PhotoController** and the **HomeController**. In which folder should you create the partial view file?

Using Partial Views

Besides knowing how to create partial views, you need to know how to use them within multiple other views in your site. You can re-use partial views by using the **Html.Partial()** and **Html.Action()** HTML helpers.

- Using HTML helpers, you can use partial views within other views in a web application:
  - To pass the same model object to a partial view from the parent view, use **Html.Partial()**
  - To pass a model object to a partial view, which is different from the parent view or of a different model class, use **Html.Action()**
- Use the **ViewBag** and **ViewData** collections to share data between the controller action, parent view, and partial view

The Html.Partial() Helper

You can use the **Html.Partial()** helper to render a partial view within another view file. MVC inserts the HTML that the partial view renders into the parent view and returns the complete webpage to the browser. The first parameter is the name of the partial view file without its file extension. The

second optional parameter is a model object for the partial view to render. This is the same model object that the parent view uses because other model objects are not available in this context.

The following lines of code call the **_Comments.cshtml** partial view without passing a model object.

Using the Html.Partial() Helper

```
@Html.Partial("_Comments")
```

Remember that you can use the **ViewBag** and **ViewData** collections to pass data between the action method and the view. A parent view will share the same **ViewBag** or **ViewData** objects with the partial view. This is a good way to share data between the controller action, view, and partial view.

The Html.Action() Helper

You can pass a model object from the parent view to a partial view when you use **Html.Partial()**. However, the parent view can only pass the same model object available in the parent view. Sometimes, you may want to use a model object different from the parent view, often of a different model class, with the partial view. In such cases, you can use the **Html.Action()** helper. This calls an action method that returns the correct partial view. The action method can create an instance of any model class.

Consider an **Article** view that displays the text and title of an **Article** object. You want to display comments at the end of this view in a partial view. You can use the **Html.Action()** helper to call an action in the **Comments** controller that returns such a partial view. The partial view can work with a collection of **Comment** objects.

In the following example, the **Html.Action()** helper calls an action in the **CommentController** controller and passes the ID of the current article model object.

Using the Html.Action() Helper

```
@Html.Action("_CommentsForArticle", "Comment", new { ArticleID = Model.ID })
```

Question: You want to display user reviews of a product to each product page. You have created a **_ReviewsForProduct** partial view. Would you use **Html.Partial()** or **Html.Action()** to render this view?

Discussion: Partial View Scenarios

Consider the following scenarios when creating a web application:

Determine the appropriate use of HTML helpers in these scenarios:

- In a product web application:
  - You want to display the ten latest comments in a new webpage when users click the **Latest Comments** link on the home page
- In a product web application:
  - You want to display the comments for an article in the **Article** view
  - You also want to display the comments for a product in the **Product** view
  - There are separate **ArticleComment** and **ProductComment** classes in your model, but they have similar properties
- In a photo sharing web application:
  - You want to display a gallery of thumbnail images on both the **AllPhotos** view and the home page **Index** view.
  - You want the **AllPhotos** view to display every **Photo** object, but you want the home page **Index** view to display only the three most recent photos uploaded

- You want to display the ten latest comments in the web application. When users click the **Latest Comments** link on the home page, they will see a new webpage with the comments displayed.
- You want to display the comments for an article on the **Article** view. You also want to display the comments for a product on the **Product** view. There are separate **ArticleComment** and **ProductComment** classes in your model, but they have similar properties.
- You have a photo sharing web application and you want to display a gallery of thumbnail images on both the **AllPhotos** view and the home page **Index** view. The **AllPhotos** view should show

every **Photo** object, but the home page **Index** view should display only the three most recent photos uploaded.

For each scenario, discuss with the class:

- Whether you need to create a view or a partial view.
- Whether you need to create a strongly-typed view or a dynamic view.
- If you decide to create a partial view, whether you need to use the **Html.Partial()** or the **Html.Action()** HTML helper in the parent view.

Lab: Developing ASP.NET MVC 4 Views

Scenario

You have been asked to add the following views to the photo sharing application:

- *A Display view for the Photo model objects*. This view will display a single photo in a large size, with the title, description, owner, and created date properties.
- *A Create view for the Photo model objects*. This view will enable users to upload a new photo to the gallery and set the title and description properties.
- *A Photo Gallery partial view*. This view will display many photos in thumbnail sizes, with the title, owner, and created date properties. This view will be used on the **All Photos** webpage to display all the photos in the application. In addition, this view will also be used on the home page to display the three most recent photos.

After adding these three views to the photo sharing application, you will also test the working of the web application.

Objectives

After completing this lab, you will be able to:

- Add Razor views to an MVC application and set properties such as scaffold and model binding.
- Write both HTML markup and C# code in a view by using Razor syntax.
- Create a partial view and use it to display re-usable markup.

Lab Setup

Estimated Time: 60 minutes

Virtual Machine: **20486B-SEA-DEV11**

User Name: **Admin**

Password: **Pa$$w0rd**

 Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

- On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.
- In the navigation pane of the **Options** dialog box, click **Package Manager**.
- Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Adding a View for Photo Display

Scenario

In this exercise, you will:

- Create a new view in the Photo Sharing web application to display single photos in large size.
- Display the properties of a photo such as title, description, and created date.

The main tasks for this exercise are as follows:

1. Add a new display view.
2. Complete the photo display view.

▶ Task 1: Add a new display view.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the Photo Sharing Application solution from the following location:
   - File location: **Allfiles (D):\Labfiles\Mod05\Starter\PhotoSharingApplication**
3. Open the PhotoController.cs code window.
4. Build the solution
5. Add a new view to the **Display** action in the **PhotoController** by using the following information:
   - Name: **Display**
   - View engine: **Razor (CSHTML)**
   - View type: **Strongly typed**
   - Model class: **Photo**
   - Scaffold template: **Empty**
   - Layout or master page: **None**

▶ Task 2: Complete the photo display view.

1. Add a title to the display view by using the **Title** property of the **Model** object.
2. Add an **H2** element to the body page to display the photo title on the page by using the **Title** property of the **Model** object.
3. Add an **<img>** tag to display the photo on the page by using the following information:
   - Width: **800**
   - src: **Empty**
4. Add the **URL.Action** helper to the **src** attribute of the **<img>** tag by using the following information:
   - Method: **Url.Action()**
   - Action name: **GetImage**
   - Controller name: **Photo**
   - Route values: **new { id=Model.PhotoID }**
5. Add a **P** element to display the **Description** property from the model by using the following information:
   - Helper: **Html.DisplayFor**
   - Lamda expression: model => model.Description
6. Add a **P** element to display the **UserName** property from the model by using the following information:

- Helpers:
  - **Html.DisplayNameFor**
  - **HtmlDisplayFor**
- Lamda expression: **model => model.UserName**

7. Add a **P** element to display the **CreatedDate** property from the model by using the following information:
   - Helpers:
     - **Html.DisplayNameFor**
     - **Html.DisplayFor**
   - Lamda expression: **model => model.CreatedDate**
8. Add a **P** element to display a link to the **Index** controller action by using the following information:
   - Helper: **HTML.ActionLink**
   - Content: **Back to List**

Note: You will create the **Index** action and view for the **PhotoController** later in this lab.

9. Save the Display.cshtml file.

Results: After completing this exercise, you will be able to add a single display view to the Photo Sharing web application and display the properties of a photo.

Exercise 2: Adding a View for New Photos

Scenario

In this exercise, you will

- Create a view to upload new photos for display in the Photo Sharing application.
- Display the properties of a photo, such as title, description, and created date.

The main tasks for this exercise are as follows:

1. Add a new create view.

2. Complete the photo create view.

▶ Task 1: Add a new create view.

1. Create a new view for the **Create** action of the **PhotoController** class by using the following information:
   - Name: **Create**
   - View: **Razor (CSHTML)**
   - View type: **Strongly Typed**
   - Model class: **Photo**
   - Scaffold template: **Empty**
   - Partial view: **None**

- Layout or master page: **None**

▶ Task 2: Complete the photo create view.

1. Add the following title to the Create view:
   - Title: **Create New Photo**
2. Add an **H2** element to the body page to display the heading as **Create New Photo**.
3. Create a form on the page by using the following information within an **@using** statement:
   - Helper: **Html.BeginForm**
   - Action: **Create**
   - Controller name: **Photo**
   - Form method: **FormMethod.Post**
   - Parameter: Pass the HTML attribute **enctype = "multipart/form-data"**
4. In the form, use the **Html.ValidationSummary** helper to render validation messages.
5. After the **ValidationSummary**, add a **P** element to display controls for the **Title** property of the model by using the following information:
   - Helpers:
     - **LabelFor**
     - **EditorFor**
     - **ValidationMessageFor**
6. After the controls for the **Title** property, add a **P** element to display a label for the **PhotoFile** property, and an **<input>** tag by using the following information:
   - Helper: **LabelFor**
   - Input type: **file**
   - Name: **Image**
7. After the **PhotoFile** controls, add a **P** element to display controls for the **Description** property of the model by using the following information:
   - Helpers:
     - **LabelFor**
     - **EditorFor**
     - **ValidationMessageFor**
8. After the **Description** controls, add a **P** element to display read-only controls for the **UserName** property of the model by using the following information:
   - Helpers:
     - **LabelFor**
     - **DisplayFor**
9. After the **UserName** controls, add a **P** element to display read-only controls for the **CreatedDate** property of the model by using the following information:
   - Helpers:
     - **LabelFor**

- **DisplayFor**

10. After the **CreatedDate** controls, add a **P** element that contains an **<input>** tag by using the following information:
    - Input type: **submit**
    - Value: **Create**
    - Add an action link to the **Index** action with the text **Back to List**.
11. Save the Create.cshtml file.

Results: After completing this exercise, you will be able to create a web application with a Razor view to display new photos.

Exercise 3: Creating and Using a Partial View

Scenario

In this exercise, you will:

- Add a gallery action to the Photo Controller.
- Add a photo gallery partial view.
- Complete the photo gallery partial view.
- Use the photo gallery partial view.

The main tasks for this exercise are as follows:

1. Add a gallery action to the Photo Controller.
2. Add a photo gallery partial view.
3. Complete the photo gallery partial view.
4. Use the photo gallery partial view.

▶ Task 1: Add a gallery action to the Photo Controller.

1. Add a new action to the PhotoController.cs file by using the following information:
    - Annotation: **ChildActionOnly**
    - Scope: **public**
    - Return Type: **ActionResult**
    - Name: **_PhotoGallery**
    - Parameter: an **Integer** called **number** with a default value of 0
2. Create a new **List** of **Photo** objects named **photos**. Add an **if** statement, to set **photos** to include all the **Photos** in the **context** object, if the **number** parameter is zero.
3. If the **number** parameter is not zero, set **photos** to list the most recent **Photo** objects. The number of **Photo** objects in the list should be the **number** attribute.
4. Pass the **photos** object to the partial view **_PhotoGallery** and return the view.
5. Save the PhotoController.cs file.

▶ Task 2: Add a photo gallery partial view.

1. Create a new partial view for the **_PhotoGallery** action in the PhotoController.cs file by using the following information:
   - Name: **_PhotoGallery**
   - View type: **Strongly Typed**
   - Model class: **Photo**
   - Scaffold template: **Empty**
   - Layout or master page: **None**
2. Create a new folder in the PhotoSharingApplication project by using the following information:
   - Parent folder: **Views**
   - Folder name: **Shared**
3. Move the _PhotoGallery.cshtml view file from the Photo folder to the Shared folder.

▶ Task 3: Complete the photo gallery partial view.

1. In the _PhotoGallery.cshtml partial view file, bind the view to an enumerable list of **Photo** model objects.
2. In the _PhotoGallery.cshtml partial view file, add a **For Each** statement that loops through all the items in the **Model**.
3. In the **For Each** statement, add an **H3** element that renders the **item.Title** property.
4. After the **H3** element, add an **if** statement that checks that the **item.PhotoFile** value is not null.
5. If the **item.PhotoFile** value is not null, render an **<img>** tag with **width 200**. Call the **UrlAction** helper to set the **src** attribute by using the following information:
   - Action: **GetImage**
   - Controller: **Photo**
   - Parameters: for the **id** parameter, pass **item.PhotoID**
6. After the **if** statement, add a **P** element, and call the **@Html.DisplayFor** helper to render the words **Created By:** followed by the value of the **item.UserName** property.
7. After the **UserName** display controls, add a **P** element, and call the **@Html.DisplayFor** helper to render the words **Created On:** followed by the value of the **item.CreatedDate** property.
8. After the **CreatedDate** display controls, call the **Html.ActionLink** helper to render a link by using the following information:
   - Link text: **Display**
   - View name: **Display**
   - Parameters: pass the **item.PhotoID** value as the **id** parameter
9. Save the _PhotoGallery.cshtml file.

▶ Task 4: Use the photo gallery partial view.

1. Modify the **Index** action in the PhotoController.cs so that no model class is passed to the **Index** view.
2. Create a view for the **Index** action in the PhotoController.cs file by using the following information:
   - Name: **Index**

- View type: **Not strongly typed**
- Layout or master page: **None**

3. In the Index.cshtml file, change the title to **All Photos**.
4. Add an **H2** element to the page body to display the heading as **All Photos**
5. Add a **P** element to add a link to the **Create** action in the **Photo** controller by using the following information:
   - Helper: **Html.ActionLink**
   - Link text: **Add a Photo**
   - Action name: **Create**
   - Controller name: **Photo**
6. Insert the **_PhotoGallery** partial view by using the following information:
   - Helper: **Html.Action**
   - Action name: **_PhotoGallery**
   - Controller name: **Photo**
7. Save the Index.cshtml file.

Results: After completing this exercise, you will be able to create a web application with a partial view to display multiple photos.

Exercise 4: Adding a Home View and Testing the Views

Scenario

In this exercise, you will create a home page that re-uses the photo gallery object, but displays only the three most recent photos.

The main tasks for this exercise are as follows:

1. Add a Controller and View for the home page.

2. Use the web application.

▶ Task 1: Add a Controller and View for the home page.

1. Add a new **Controller** to the home page by using the following information:
   - Controller name: **HomeController**
   - Template: **Empty MVC Controller**
2. Add a new view to the **Index** action in **HomeController** by using the following information:
   - View name: **Index**
   - View type: **Not strongly typed**
   - Layout or master page: **None**
3. Change the title of the page to **Welcome to Adventure Works Photo Sharing**.
4. Add the following text to the home page:
   - **Welcome to Adventure Works Photo Sharing! Use this site to share your adventures.**

5. Add an **H2** element to display the heading as **Latest Photos**.
6. Insert the **_PhotoGallery** partial view by using the following information:
   - Helper: **Html.Action**
   - Action name: **_PhotoGallery**
   - Controller name: **Photo**
   - Parameters: for the **number** parameter, pass the value **3**
7. Save the Index.cshtml file.

▶ Task 2: Use the web application.

1. Start the Photo Sharing web application with debugging.
2. Verify the number of photos displayed on the home page.
3. Display a photo of your choice to verify whether the display shows the required information.
4. Verify the number of photos displayed on the **All Photos** page.
5. Add a new photo of your choice to the application by using the following information:
   - Title: **My First Photo**
   - Description: **This is the first test of the Create photo view**.
   - File path: **Allfiles (D):\Labfiles\Mod05\SamplePhotos**
6. Close Internet Explorer and Microsoft Visual Studio.

Results: After completing this exercise, you will be able to create a web application in which users can upload and view the photos.

Question: How can you improve the accessibility of the HTML that your photo views render?

Question: In the lab, how did you ensure that the **Create** view for **Photo** model objects could upload photo files when the user clicked the **Create** button?

Module Review and Takeaways

In this module, you saw how to build the user interface for your web application by creating views with the Razor view engine. Views can display properties from a model class and enable users to create, read, update, and delete objects of that model class. Some views display many objects by looping through a collection of model objects. The HTML Helper methods that are built into ASP.NET MVC 4 facilitate the rendering of displays, controls, and forms, and return validation messages to users. Partial views can be re-used several times in your web application to render similar sections of HTML on multiple pages.

Best Practice: Use Razor comments, declared with the **@* *@** delimiters, throughout your Razor views to help explain the view code to other developers in your team.

Best Practice: Only use **@:** and **<text>** tags when Razor misinterprets content as code. Razor has sophisticated logic for distinguishing content from code, so this is rarely necessary.

Best Practice: Use strongly-typed views wherever possible because Visual Studio helps you to write correct code by displaying IntelliSense feedback.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
When a controller tries to access a partial view, an exception is thrown.	

Review Question(s)

Question: You want to display the name of the **Comment.Subject** property in an MVC view that renders an Edit form for comments. You want the label Edit Subject to appear to the left of a text box so that a user can edit the **Subject** value. Which HTML helper should you use to render the field name?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
If your Razor generates errors by wrongly interpreting content as server-side code, this indicates that you have explicitly declared the content by using the **@* *@** delimiters.	

Module 06

Testing and Debugging ASP.NET MVC 4 Web Applications

Contents:

Module Overview

Software systems such as web applications are complex and require multiple components, often written by different developers, to work together. Incorrect assumptions, inaccurate understanding, coding errors, and many other sources can create bugs that result in exceptions or unexpected behavior. To improve the quality of your web application and create a satisfying user experience, you must identify bugs from any source and eliminate them. Traditionally, most testing has been performed at the end of a development project. However, it has recently become widely accepted that testing throughout the project life cycle improves quality and ensures that there are no bugs in production software. You need to understand how to run tests on small components of your web application to ensure that they function as expected before you assemble them into a complete web application. You also need to know how to use the tools available in Microsoft Visual Studio 2012 to debug and trace exceptions when they arise.

Objectives

After completing this module, you will be able to:

- Run unit tests against the Model–View–Controller (MVC) components, such as model classes, and locate potential bugs.
- Build an MVC application that handles exceptions smoothly and robustly.

Lesson 1
Unit Testing MVC Components

A unit test is a procedure that instantiates a component that you have written, calls one aspect of the functionalities of the component, and checks that the component responds correctly according to the design. In object-oriented programming, unit tests usually instantiate a class and call one of its methods. In an MVC web application, unit tests can instantiate model classes or controllers and call their methods and actions. As a web developer, you need to know how to create a testing project and add tests to the testing project. The tests check individual aspects of your code without relying on database servers, network connections, and other infrastructure. This is because unit tests focus on the code you write.

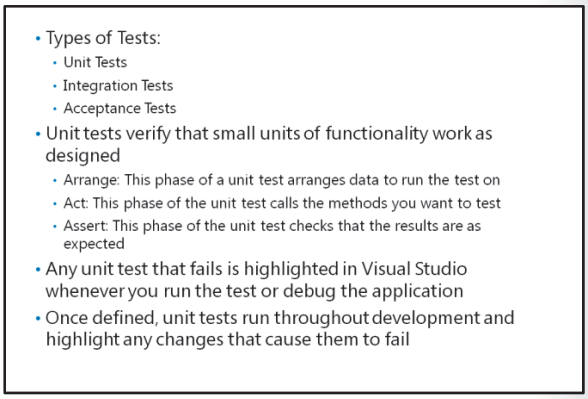
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of unit tests to improve the quality of code you write.
- List the principles of Test Driven Development (TDD).
- Describe how loosely-coupled components increase the testability and maintainability of your application.
- Write unit tests against model classes, controllers, and views in an MVC web application.
- Describe how Inversion of Control (IoC) containers can instantiate controllers and model classes.
- Run unit tests in Visual Studio 2012.
- Use a mocking tool to automate mock creation in unit tests.

Why Perform Unit Tests?

There are three types of tests you can use to identify bugs in your application:



- *Unit Tests:* Unit tests check small aspects of functionality. For example, a unit test may verify the return type of a single method. By defining many unit tests for your project, you can ensure they cover all functional aspects of your web application.
- *Integration Tests:* Integration tests check how two or more components work together. They can be used to check how two or more classes interact with each other. Integration tests can also be used to check how the entire web application, including the database and external web services, works to deliver content.
- *Acceptance Tests:* Acceptance tests focus on a functional or technical requirement that must work for the stakeholders to accept the application. Similar to integration tests, acceptance tests usually test multiple components working together.

Note: Unit tests are important because they can be defined early in development. Integration and acceptance tests are usually run later, when several components are approaching completion.

What Is a Unit Test?

A unit test is a procedure that verifies a specific aspect of functionality. Multiple unit tests can be performed for a single class and even for a single method in a class. For example, you can write a test that checks that the **Validate** method always returns a Boolean value. You might write a second test that checks that when you send a string to the **Validate** method, the method returns a **true** value. You can assemble many unit tests into a single class called a test fixture. Often, a test fixture contains all the tests for a specific class. For example, **ProductTestsFixture** may include all tests that check methods in the **Product** class.

A single unit test usually consists of code that runs in three phases:

1. *Arrange*. In this phase, the test creates an instance of the class that it will test. It also assigns any required properties and creates any required objects to complete the test. Only properties and objects that are essential to the test are created.
2. *Act*. In this phase, the test runs the functionality that it must check. Usually, in the Act phase, the test calls a single procedure and stores the result in a variable.
3. *Assert*. In this phase, the test checks the result against the expected result. If the result matches what was expected, the test passes. Otherwise, the test fails.

How Do Unit Test Help Diagnose Bugs?

Because unit tests check a small and specific aspect of code, it is easy to diagnose the problem when the tests fail. Unit tests usually work with a single class and isolate the class from other classes wherever possible. If other classes are essential, the smallest number of classes are created in the Arrange phase. This approach enables you to fix issues rapidly because the number of potential sources of a bug is small.

Unit tests should check the code that you write and not any infrastructure that the production system will rely on. For example, unit tests should run without connecting to a real database or web service because network problems or service outages may cause a failure. Using this approach, you can distinguish bugs that arise from code, which must be fixed by altering code, from the bugs that arise from infrastructure failures, which must be fixed by changing hardware, reconfiguring web servers, reconfiguring connection strings, or making other configuration changes.

Automated Unit Testing

It is important to understand that unit tests, after they are defined, can be rerun quickly and easily throughout the rest of the project life cycle. In fact, Microsoft Visual Studio reruns tests automatically whenever you run a project with debugging. You can also manually initiate tests any time.

This is important because new code can cause bugs at any stage of the development process. As an example, consider a project that proceeds through three phases. A unit test defined for phase 1 helps highlight problems in phases 2 and 3 that might otherwise go unnoticed:

1. *Phase 1*. The **ProductController** class is defined and the **Test_Index_Action** unit test is written. The test checks that the **Index** action works with an integer parameter. When you call the **Index** action with an integer, the action returns a collection that includes the same number of **Product** objects. The test passes.
2. *Phase 2*. A developer modifies the **Index** action so that it returns a partial view. The developer renames the action **_Index** to conform to the team's partial view naming convention. The test fails because the name has changed. The developer modifies the test so that it calls the renamed action and the test passes.

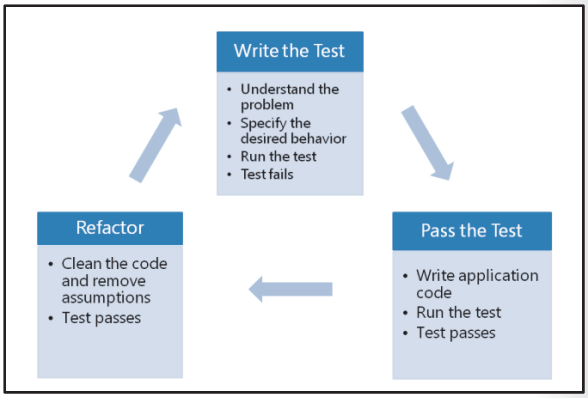
3. *Phase 3*. A developer modifies the **Index** action by writing a different Language Integrated Query (LINQ) to implement a new functional requirement. However, the developer makes a mistake in the LINQ syntax. The **Index** action now returns zero products regardless of the integer that is passed as a parameter. The test fails.

In Phase 2, the test failure arose because the action name had changed, not because of a mistake in the action code. The solution was to change the name of the action called in the test method, but this can remind developers to alter calls to renamed actions throughout the solution.

In Phase 3, the test failure arose because the developer misunderstood LINQ or made a typing error in the LINQ syntax. The unit test highlighted the error as soon as it arose. This helps the developer focus on a solution to resolve the test failure and ensure that new functionality does not cause failures in code written earlier in the project.

Question: Board members want you to ensure that your web application correctly calculates sales tax on every product in the catalog. Is this an example of a unit test, an integration test, or an acceptance test?

Principles of Test Driven Development



You can use unit tests in any development methodology, including waterfall projects, iterative projects, and Agile projects to spot potential bugs and improve the quality of the final application. Any project can benefit from unit testing, regardless of the development methodology used in the project. However, a specific development methodology, called Test Driven Development (TDD), places unit testing at the center of working practices. TDD is often described as a separate development methodology. Some authors consider it to be a core principle of the Extreme Programming methodology.

Write the Test, Pass the Test, Refactor

In TDD, developers repeat the following short cycle to implement a functional requirement:

1. *Write the Test*. The developer starts by creating and coding the unit test. This step requires a full understanding of the functional requirement, which can be obtained from use cases or user stories. Because the developer has not written any code in the application, the test fails.
2. *Pass the Test*. The developer writes some quick and simple code in the application so that it passes the test. During the first iteration, this code is frequently inelegant and may include false assumptions such as hardcoded values.
3. *Refactor*. The developer cleans up the application code, removes duplicate code, removes hardcoded values, improves readability, and makes other technical improvements. The developer reruns the test to ensure that refactoring has not caused a failure.

The cycle is then repeated. In each iteration, the developer adds a small new element of the final functionality with a corresponding test.

It is important that the code in your tests is treated as production code. It should be well thought out and easy to read so that other developers can understand the test and quickly diagnose any test failures.

Test Driven Development Principles

TDD is different from traditional approaches to application development. To use TDD effectively, you must understand its fundamental principles:

- *Write tests before code*. In the TDD development cycle, the test is created in step one before any code in the application. This means the test must fail the first time it is run. You can understand the test as a specification for the functionality you are building. By writing the test first, you ensure that you begin with a thorough understanding of the problem you are trying to solve.
- *Move in small steps*. By breaking a large application down into small elements of functionality, you can improve developer productivity. You can do this by giving a developer a small problem to solve in each iteration. The developer can solve the simple problem quickly and understand all the possible circumstances in which their code runs.
- *Only write enough code to pass the test*. In each iteration, do not be tempted to add extra code that is not related to the test. For example, if you know that users will call an action with other parameters than the ones in the test, do not write code to handle these parameters. Instead, during the next iteration, write a second test for those parameters. Then write and refactor that code.

Developers can refer to tests as examples of how to use a class or method. This can increase developer productivity because developers can view the code that demonstrates the method or class.

Note: In the lab, you will use some TDD principals as you develop the solution. However, the lab does not conform to a strict reading of the TDD methods.

Question: You have written a controller for the **Comment** model class. You write a unit test that checks that the **Index** action returns a collection of **Comment** objects. Have you conformed to TDD principles?

Writing Loosely Coupled MVC Components

A loosely coupled application is one in which each component requires few or no details of the other components of the system. In object-oriented programming, for example, two classes can be described as loosely-coupled if one class can call methods on the other class without any code that is specific to the other class. When system components are loosely-coupled in this manner, it is easy to replace a class with another implementation of the same functionality. Loosely coupled components are essential for thorough unit testing because classes that deal with real data, such as data from a database, can be easily be replaced with classes that deal with test data.

- Loose coupling means that each component in a system requires few or no internal details of the other components in the system
- A loosely-coupled application is easy to test because it is easier to replace a fully functional instance of a class with a simplified instance that is specifically designed for the test
- Loose coupling makes it easier to replace simple components with more sophisticated components

Using Loose Coupling in Tests

A loosely-coupled application is easy to test because you can make tests simpler by replacing a fully functional instance of a class with a simplified instance that is specifically designed for the test. Replacing classes in tests in this manner can only be done when components are loosely coupled. Replacement instances used for unit tests are known as test doubles or fakes. A test double or fake includes just enough code and properties to pass the relevant test and prove the functionality.

Other Benefits of Loose Coupling

Loose coupling has other benefits besides testing. Loose coupling makes it easier to replace simple components with more sophisticated ones—for example, in version one of an application you might write a class that calculates a simple arithmetical mean. In version two, you might replace this class with a new class that calculates a weighted mean based on a more complex algorithm. If the components are loosely coupled, you can perform this replacement without modifying any code outside of the averaging classes.

Using Interfaces for Loose Coupling

In object-oriented programming, an interface defines a set of properties and methods. Any class that implements that interface must implement all the properties and methods it defines as a minimum. This creates loose coupling because you need not specify a class in code. Instead, you can specify any implementation of a particular interface.

Loose Coupling in an MVC Web Application

To show how loose coupling can be used in an MVC web application, consider the following scenario:

You are writing an MVC web application that displays a product catalog and enables users to click a product to view the full details. On the product details page, users can view and add reviews of that product. You want to ensure during testing that the **Product.Reviews** property, which is a collection of **Review** objects, only includes reviews with the right **ProductID** value.

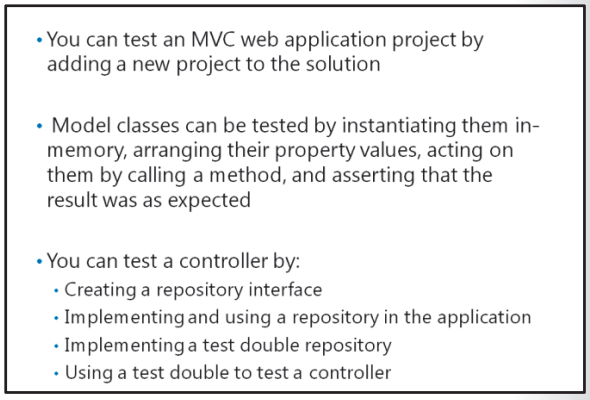
Recall that unit tests should not rely on infrastructure such as network connections or database servers, but only on test code. By contrast, in production, the application will obtain all product reviews from a database. To satisfy the needs of testing and production, you can build the following components:

- An **IProduct** interface. This includes a definition of the **Reviews** property, together with other properties and methods. This interface is known as a repository.
- A **Product** class. This is the implementation of the **IProduct** repository that the application uses in production. When the user calls the **Reviews** property, all the reviews for the current product are obtained from the database.
- A **TestProduct** class. This is the test double or fake implementation of the **IProduct** repository that the test uses. The test sets up a **TestProduct** object and fake reviews with different **ProductID** values. The test calls the **TestProduct.Reviews** property and checks that only the right reviews are returned.

Notice that, in the test, the **TestProduct** double uses in-memory data set up during the *Arrange* phase. It does not query the database. Therefore, you can test your code without relying on the network connection to the database or the database server itself. This approach also ensures that the test runs quickly, which is important because slow tests discourage developers from running tests regularly.

Question: In a test, you create a fake collection of **BlogEntry** objects. Is this an example of an interface or a test double?

Writing Unit Tests for MVC Components



The Microsoft ASP.NET MVC 4 programming model is easy to integrate with the principles of unit testing and TDD because of its separation of concerns into model, controllers, and views. Models are simple to test because they are independent classes that you can instantiate and configure during the Arrange phase of a test. Controllers are simple classes that you can test, but it is complex to test controllers with in-memory data, rather than using a database. To test controllers with in-memory data, you create a test double class, also known as a fake repository. Objects of this class can be populated with test data in memory without querying a database. You need to understand how to write test doubles and how to write a typical test for MVC classes.

Add and Configure a Test Project

In Microsoft Visual Studio 2012, you can test an MVC web application project by adding a new project to the solution, based on the **Unit Test Project** template. You must add references to the test project so that test code can access classes in the MVC web application project. You will also need a reference to **System.Web.Mvc** and other namespaces from the Microsoft .NET Framework.

In a Visual Studio test project, test fixtures are classes marked with the **[TestClass]** annotation. Unit tests are methods marked with the **[TestMethod]** annotation. Unit tests usually return **void** but call a method of the **Assert** class, such as **Assert.AreEqual()** to check results in the test Assert phase.

Note: There are many other test frameworks available for MVC web applications and you can choose the one you prefer. Many of these frameworks can be added by using the NuGet package manager available in Visual Studio. If you use Visual Studio Express, you cannot use the Visual Studio **Unit Test Project** template, but you can add other testing frameworks by using NuGet.

Test Model Classes and Business Logic

In MVC, model classes do not depend on other components or infrastructure. You can easily instantiate them in-memory, arrange their property values, act on them by calling a method, and assert that the result was as expected. Sometimes, you create business logic in the model classes, in which case the *Act* phase involves calling a method on the model class itself. If, by contrast, you have created a separate business logic layer, code in the *Act* phase must call a method on the business object class and pass a model class.

The code in this test creates an instance of the **Product** model class and sets its **Type** property. The test checks that in these circumstances, the **GetAccessories()** method returns a collection of **BikeAccessory** objects.

Testing a Model Class

```
 [TestMethod]
public void Test_Product_Reviews()
{
   //Arrange phase
   Product testProduct = new Product();
   testProduct.Type = "CompleteBike";
   //Act phase
   var result = testProduct.GetAccessories();
```

```
    //Assert phase
    Assert.AreEqual(typeof(IEnumerable(BikeAccessory)), result.GetType());
}
```

Create Repository Interfaces

A repository is an interface that defines properties and methods that MVC can use to store data in a web application. It is common to create a single repository interface for your entire web application.

In the following code example, the **IWebStoreContext** interface is a repository that defines methods such as **Add()** and **Delete()**, and properties such as **Products**.

A Repository Interface

```
public interface IWebStoreContext
{
    IQueryable<Product> Products { get; }
    T Add<T>(T entity) where T : class;
    Product FindProductById(int ID);
    T Delete<T>(T entity) where T : class;
}
```

Implement and Use a Repository in the Application

The repository interface defines methods for interacting with data but does not fix how that data will be set and stored. You must provide two implementations of the repository:

- An implementation of the repository for use in production and development. This implementation will use data from the database or some other storage mechanism.
- An implementation of the repository for use in tests. This implementation will use data from the memory set during the Arrange phase of each test.

The following code example shows how to implement the repository interface in your MVC project for use in production and development.

Implementing a Repository in the MVC Project

```
public class WebStoreContext : DbContext, IWebStoreContext
{
    public DbSet<Product> Products { get; set; }
    IQueryable<Product> IWebStoreContext.Products { get { return Photos; } }
    T IWebStoreContext.Add<T>(T entity)
    {
        return Set<T>().Add(entity);
    }
    Product IWebStoreContext.FindProductById(int ID)
    {
        return Set<Product>().Find(ID);
    }
    T IWebStoreContext.Delete<T>(T entity)
    {
        return Set<T>().Remove(entity);
    }
}
```

Note that the **WebStoreContext** class still inherits from the Entity Framework **DbContext** class but also implements the **IWebStoreContext** interface that you just created. In fact, the interface methods such as **Add()**, **Delete()**, and **FindProductById()** just wrap methods from the **DbContext** class such as **Remove()**.

Implement a Repository Test Double

The second implementation of the repository interface is the implementation that you will use in unit tests. This implementation uses in-memory data and a keyed collection of objects to function just like an Entity Framework context, but avoids working with a database.

The following code example shows how to implement a repository class for tests.

Implementing a Fake Repository

```
class FakeWebStoreContext : IWebStoreContext
{
   //This object is a keyed collection we use to mock an
   //entity framework context in memory
   SetMap _map = new SetMap();
   public IQueryable<Product> Products
   {
       get { return _map.Get<Product>().AsQueryable(); }
       set { _map.Use<Product>(value); }
   }
   public T Add<T>(T entity) where T : class
   {
       _map.Get<T>().Add(entity);
       return entity;
   }
   public Product FindProductById(int ID)
   {
       Product item = (from p in this.Products
                   where p.ProductID == ID
                   select p).First();
       return item;
   }
   public T Delete<T>(T entity) where T : class
   {
       _map.Get<T>().Remove(entity);
       return entity;
   }
   class SetMap : KeyedCollection<Type, object>
   {
       public HashSet<T> Use<T>(IEnumerable<T> sourceData)
       {
           var set = new HashSet<T>(sourceData);
           if (Contains(typeof(T)))
           {
               Remove(typeof(T));
           }
           Add(set);
           return set;
       }
       public HashSet<T> Get<T>() { return (HashSet<T>) this[typeof(T)]; }
       protected override Type GetKeyForItem(object item)
       {
           return item.GetType().GetGenericArguments().Single();
       }
   }
}
```

Using the Test Double to Test a Controller

After you have implemented a test double class for the repository, you can use it to test a controller in a unit test. In the Arrange phase of the test, create a new object from the test double class and pass it to the controller constructor. The controller uses this object for its Entity Framework context during the test Act phase. This enables you to check results.

The following code example shows how to test a controller action by using a test double for the Entity Framework context.

Using a Test Double to Test a Controller

```
 [TestMethod]
public void Test_Index_Model_Type()
{
   var context = new FakeWebStoreContext();
   context.Products = new[] {
      new Product(),
      new Product (),
      new Product ()
   }.AsQueryable();
   var controller = new ProductController(context);
   var result = controller.Index() as ViewResult;
   Assert.AreEqual(typeof(List<Product>), result.Model.GetType());
}
```

Note that the test creates a new instance of the **FakeWebStoreContext** class and adds three **Product** objects to it. In this case, you can run the test without setting any properties on the **Product** objects. This context is passed to the **ProductController** constructor and the test proceeds to the Act and Assert phases.

Question: For what purpose would you use a fake repository when you write unit tests against an MVC controller?

Specifying the Correct Context

When you use test doubles for unit testing MVC controllers, you must ensure that the controller uses the correct context in two different scenarios: unit tests, and at all other times. If you use the test double production code, for example, no data will be stored in the database. If you use the actual Entity Framework context for unit testing, all the tests will fail. The following table gives further details.

To set the correct context while testing:

- Use a test double context in unit tests
- Use an Entity Framework context at other times
- Use constructors to specify the context
- Use IoC containers to specify the context

Scenario	Context
In unit tests	In a unit test, you should test code without relying on underlying infrastructure such as databases. You can use a test double class that behaves like an Entity Framework context, but uses in-memory data.
At other times	When you are not unit testing the code, you use an actual Entity Framework context in your controller code, which interacts with a database.

Using Constructors to Specify a Repository

You can specify the correct repository to use in a test by using controller constructors. During a unit test, call a controller constructor that takes an instance of the repository interface as a parameter. At other times, the MVC controller factory will call the controller constructor without any parameters.

The following code example shows the constructors in an MVC controller that you can use for production and unit testing.

Using Constructors to Specify Repositories

```
public class ProductController : Controller
{
   private IWebStoreContext context;
   //The parameterless version of the constructor is used by the MVC controller factory
   public ProductController()
   {
      //Instantiate an actual Entity Framework context
      context = new WebStoreContext();
   }
   //This constructor is used by unit tests. They pass a test double context
   public ProductController(IWebStoreContext Context)
   {
       //Use the context passed to the constructor
       context = Context;
   }
   //Add action methods here
}
```

Using Inversion of Control Containers

You can also ensure that the correct repository class is used in each context by using an Inversion of Control (IoC) container. An IoC container is a framework that you can add to your application that instantiates classes. You must configure the IoC container so that it can identify the kind of object to create when the application expects a specific interface. For example, when code is used to create a new instance of the **IWebStoreContext** interface, the IoC container checks its configuration and finds that it should instantiate the **WebStoreContext** class, which is an Entity Framework context that implements the **IWebStoreContext** interface.

 **Note:** IoC frameworks are also known as Dependency Injection (DI) frameworks because they inject classes that implement the correct interfaces into class constructors.

By using an IoC container, you avoid the need to create two constructors for controller classes. During tests, you can instantiate the test double and pass it to the controller constructor. At other times, the IoC container instantiates the actual Entity Framework context and passes it to the controller constructor.

 **Note:** There are many IoC containers designed for use with ASP.NET MVC web applications. These replace certain components of the MVC framework to support IoC. For example, they replace the default controller factory with a factory that instantiates controllers with the right context class. Many IoC containers are available in the NuGet package library, such as NinJect and StructureMap.

Question: Which approach is more loosely-coupled: using constructors to specify the context, or using IoC containers to specify the context?

Demonstration: How to Run Unit Tests

In this demonstration, you will see how to add a new test project to your ASP.NET MVC 4 solution, and how to create and run a simple unit test against the home page controller.

Demonstration Steps

1. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, right-click **Solution 'OperasWebSite' (1 project)**, point to **Add**, and then click **New Project**.
2. In the navigation pane of the **Add New Project** dialog box, under Installed, expand **Visual C#**, and then click **Test**.
3. In the result pane of the **Add New Project** dialog box, click **Unit Test Project**, in the **Name** box, type **OperasWebSiteTests**, and then click **OK**.
4. In the Solution Explorer pane, under OperasWebSiteTests, right-click **References**, and then click **Add Reference**.
5. In the navigation pane of the **Reference Manager - OperasWebSiteTests** dialog box, click **Solution**.
6. In the **Name** column of the result pane, click **OperasWebSite**, select the check box corresponding to OperasWebSite, and then click **OK**.
7. In the Solution Explorer pane, under OperasWebSiteTests, right-click **References**, and then click **Add Reference**.
8. In the navigation pane of the **Reference Manager - OperasWebSiteTests** dialog box, click **Assemblies**, and then click **Extensions**.
9. In the **Name** column of the result pane, click **System.Web.Mvc** with version number **4.0.0.0**, select the corresponding check box, and then click **OK**.
10. In the Solution Explorer pane, under OperasWebSiteTests, right-click **UnitTest1.cs**, and then click **Rename**.
11. In the Solution Explorer pane, replace **UnitTest1** with **HomeControllerTests.cs**, and then press Enter.
12. In the **Microsoft Visual Studio** dialog box, click **Yes**.
13. In the HomeControllerTests.cs code window, locate the following code.

```
public void TestMethod1()
```

14. Replace the code with the following code.

```
public void Test_Index_Return_View()
```

15. Place the mouse cursor at the end of the Microsoft.VisualStudio.TestTools.UnitTesting namespace, press Enter, and then type the following code.

```
using System.Web.Mvc;
using OperasWebSite.Controllers;
using OperasWebSite.Models;
```

16. In the **Test_Index_Return_View** code block, press Enter, and then type the following code.

```
HomeController controller =
   new HomeController();
var result = controller.Index()
   as ViewResult;
Assert.AreEqual("WrongName",
   result.ViewName);
```

Note: This test is created to show the students a failing test.

17. On the **TEST** menu of the **OperasWebSite - Microsoft Visual Studio** window, point to **Run**, and then click **All Tests**.
18. In the Failed Tests (1) section of the Test Explorer pane, note that **Test_Index_Return_View** is listed.
19. In the Test Explorer pane, click **Test_Index_Return_View**.
20. At the lower part of the Test Explorer pane, drag the separator upward, and then view the test results.
21. In the Test Explorer pane, click the **Close** button.
22. In the Solution Explorer pane, under OperasWebSiteTests, click **HomeControllerTests.cs**.
23. In the HomeControllerTests.cs code window, locate the following code.

```
Assert.AreEqual("WrongName",
   result.ViewName);
```

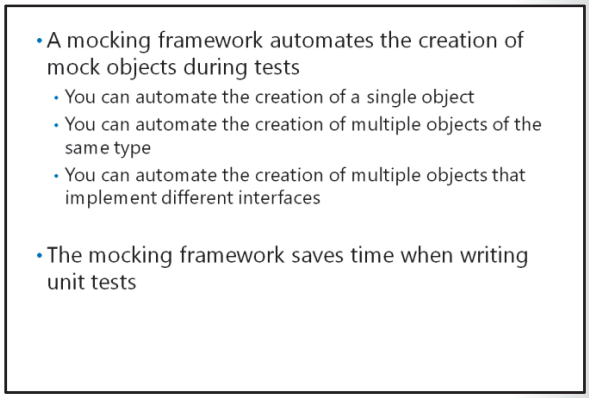
24. Replace the code with the following code.

```
Assert.AreEqual("Index",
   result.ViewName);.
```

25. On the **TEST** menu of the **OperasWebSite - Microsoft Visual Studio** window, point to **Run**, and then click **All Tests**.
26. In the Passed Tests (1) section of the Test Explorer pane, note that **Test_Index_Return_View** is listed.
27. In the Test Explorer pane, click **Test_Index_Return_View**, and then, in the lower part of the Test Explorer pane, view the test results.
28. In the Test Explorer pane, click the **Close** button.
29. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Using Mocking Frameworks

When you write a unit test, you must, during the Arrange phase, create test data on which the test will run. By using a test double or mock object to supply test data, you can test your code in isolation from other classes and from the infrastructure elements such as databases and network connections on which the application will rely. You can create test doubles by manually coding their instantiation, setting their properties, and populating collections. Such test doubles are known as manual mock objects.



Alternatively, instead of creating mock objects manually with your own code, you can use a mocking framework to automate this work. A mocking framework automatically creates mock object by using the interfaces that you specify. In the case of IoC containers, there are many mocking frameworks that can be used in testing MVC web applications. You can find many mocking frameworks by using the NuGet package manager.

 **Note:** Mocking frameworks for ASP.NET MVC 4 include Moq, RhinoMocks, and NSubstitute. All these frameworks are available in NuGet and there are alternatives. Choose the framework that best suits your testing needs.

Why Use a Mocking Framework?

There are many situations in which mocking frameworks can significantly ease unit testing. Even for simple tests, mocking frameworks reduce the amount of setup code that you have to develop. After you become familiar with the mocking framework that you choose, and have learned how to write arrangement code for it, you will begin to save time. In more complex situations, such as the following, mocking frameworks have great advantages:

- *Creating multiple mock objects of a single type*. You should try to keep each unit test as simple as possible, but inevitably, some tests require many mock objects to work with. For example, if you are testing a method that underlies paging functionality in your web application, you must create enough mock objects to populate multiple pages. Mocking frameworks can automate the creation of multiple mock objects of the same type.
- *Mocking objects with multiple interfaces*. In some tests, where there are several dependencies between classes of different types, you must create many mock objects of different classes. In such situations, it is tedious to manually code many mock objects. Mocking frameworks can help by automatically generating the objects from the interfaces that you specify.

In each unit test, you are interested in a small and specific area of functionality. Some properties and methods of an interface will be crucial to your test, while others will be irrelevant. A mocking framework enables you to specify irrelevant properties and methods in a given test. When the framework creates a mock object for your test, it creates stubs for the irrelevant properties and methods. A stub is a simple implementation with little code. In this way, the mocking framework frees you from having to implement all the requirements of the interface laboriously.

Question: What is the difference between an IoC container and a mocking framework?

Lesson 2
Implementing an Exception Handling Strategy

Unexpected events are likely to occur from time to time in any complex system, including MVC web applications. Occasionally, such unexpected run-time events cause an error. When this happens, ASP.NET or the .NET Framework generates an exception, which is an object that you can use to diagnose and resolve the error. The exception contains information that you can use to diagnose the problem. Exceptions that are not handled in your code will cause the web application to halt and an error message to be displayed to the user. As a web developer, you need to know how to detect, handle, and raise exceptions, and identify the cause of the problem. Visual Studio provides a broad range of tools for debugging exceptions and improving the robustness of your code. You can also log exceptions to databases and other stores, and use the .NET Framework code contracts to reduce the frequency of exceptions.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to raise and catch exceptions.
- Explain how to configure exception handling by using Web.config.
- Explain how to trace exceptions by using the IntelliTrace tool.
- Describe how to log exceptions.
- Describe the health monitoring options for an MVC 4 web application.

Raising and Catching Exceptions

An error is an unexpected run-time event that prevents an application from completing an operation. When a line of code causes an error, ASP.NET or the common language runtime creates an exception. This exception is an object of a class that inherits from the **System.Exception** base class. There are many exception classes. Often the object class identifies what went wrong. For example, if there is an **ArgumentNullException**, it indicates that a **null** value was sent to a method that does not accept a **null** value as an argument.

- The most common method to catch an exception is to use the **try/catch** block
- You can also override the OnException method
- You can also catch exceptions by using the **[HandleError]** annotation

```
[HandleError(ExceptionType=
  typeof(NotImplementedException),
  View="NotImplemented")]
[HandleError]
public ActionResult Index()
{
  //Place action code here
}
```

Unhandled Exceptions

When an exception is not explicitly handled by an application, the application stops and the user sees an error message. In ASP.NET MVC applications, this error message is in the form of a webpage. You can override ASP.NET default error pages to display your own error information to users.

If an unhandled exception arises while you are debugging the application in Visual Studio, execution breaks on the line that generated the exception. You can use the Visual Studio debugging tools to investigate what went wrong, isolate the problem, and debug your code.

Sometimes, you may also want to raise your own exceptions as a form of communication between parts of your application. For example, consider an application that displays products on a website. In the repository, you implement a **FindProduct** method that returns the product with a name that matches the

search string that is passed as an attribute. If the **FindProduct** method does not locate a product with the specified name, you may wish to raise an exception. This approach enables code in the controller, which calls the **FindProduct** method, to handle the event that no products are found. You can use the **throw** keyword to raise errors in C# code.

Catching Errors with Try/Catch Blocks

The most familiar way to catch an exception, which works in any .NET Framework code, is to use the **try/catch** block. Code in the **try** block is run. If any of that code generates an exception, the type of exception is checked against the type declared in the **catch** block. If the type matches, or is of a type derived from the type declared in the **catch** block, the code in the **catch** block runs. You can use the code in the **catch** block to obtain information about what went wrong and resolve the error condition.

The following code example catches any **ArgumentNullException** objects.

A Simple Try/Catch Block

```
try
{
   Product product = FindProductFromComment(comment);
}
catch (ArgumentNullException ex)
{
   Console.WriteLine("The comment object was null. Error message: " + ex.Message);
}
```

In MVC controllers, there are two methods to catch exceptions—the **OnException** method and the **[HandleError]** annotation.

Catching Controller Exceptions in OnException

The **OnException** method is defined on the base **Controller** class, so you can override it in any MVC controller. When you override the **OnException** method in a custom controller, MVC runs the method whenever an exception arises and is not handled in a **try/catch** block. Using this approach, you can handle errors anywhere in your controller without adding many **try/catch** blocks throughout the code.

In the **OnException** method, you can catch specific types of exceptions by checking the **ExceptionContext.Exception** property. Your code must set the **context.Result** property to display a view to the user. If you do not set this property, the browser displays a blank webpage to the user.

In the following code example, the **OnException** method is used to catch **InvalidOperationException** objects.

Using OnException to Catch Controller Errors

```
protected override void OnException(ExceptionContext context)
{
   // Catch invalid operation exception
   if (filterContext.Exception is InvalidOperationException)
   {
      var model = new HandleErrorInfo(context.Exception, controllerName, actionName);
      var result = new ViewResult
         {
            ViewName = "Error",
            ViewData = new ViewDataDictionary<HandleErrorInfo>(model),
            //Save pass the current Temp Data to the Error view, because it often
contains
            //diagnostic information
            TempData = context.Controller.TempData
         };
   context.Result = result;
   }
}
```

Catching Controller Exceptions with HandleError

Using the **[HandleError]** annotation, you can use an attribute on individual action methods or on the controller class itself, in which case, the **[HandleError]** annotation catches errors in any action. When you use the **[HandleError]** annotation without properties, all exceptions are caught and sent to an MVC view called **Error.cshtml**. This default error view should be created in the **/View/Shared** folder.

You can use two properties to control the behavior of **[HandleError]**:

- *ExceptionType*. By default, the **[HandleError]** annotation catches all exceptions. If you set a more specific type with the **ExceptionType** property, only exceptions of that specific type will be caught.
- *View*. By default, the **[HandleError]** annotation forwards users to the **Error.cshtml** view. You can specify another view file by using the **View** property. By setting both the **ExceptionType** and **View** properties, you can divert exceptions of specific types to views designed specifically to display them.

In the code example, the first **[HandleError]** annotation forwards **NotImplementedException** errors to the **NotImplemented** view. All other errors are forwarded to the default error view.

Using the HandleError Annotation

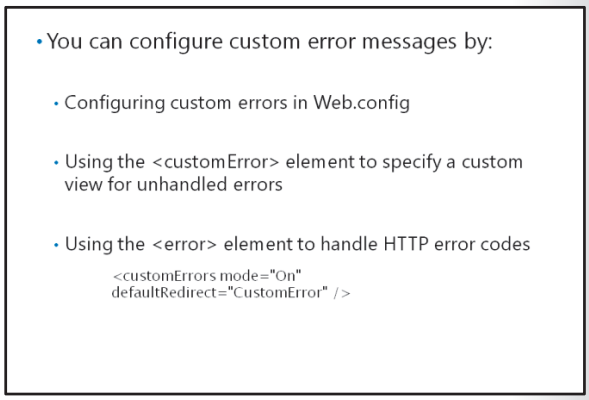
```
 [HandleError(ExceptionType=typeof(NotImplementedException), View="NotImplemented")]
[HandleError]
public ActionResult Index()
{
   //Place action code here
}
```

Question: You are using the **[HandleError]** annotation to catch exceptions in your web application. However, you realize that **[HandleError]** catches all exceptions, making it difficult for you to isolate a specific issue. What can you do to narrow down the exceptions that the **[HandleError]** annotation is catching?

Configuring Exception Handling

ASP.NET applications display a default error page to users when an unhandled exception occurs in your MVC web application. There are two reasons why you may want to change the default error page:

- You want to display a branded and styled error page that has the same layout as other pages in the application and includes the company logo.
- You want to avoid drawing attention to the technology that powers the site. By letting malicious users know that your site runs ASP.NET MVC, you may encourage them to attack it by using methods that have worked in the past.



To configure a custom error page for your application, use the top-level Web.config file. The custom error page will be displayed when an exception in not handled by any **try/catch** block or **[HandleError]** attribute.

Custom Error Modes

To configure custom errors, add the **<customErrors>** element to the Web.config file. This element must be a child of the **<system.web>** element. Use the following attributes:

- **mode:** The mode attribute can be set to one of three values:
  - **Off:** Unhandled errors are displayed on the default ASP.NET error page.
  - **On:** Unhandled errors are displayed in the page you specify in the **defaultRedirect** attribute.
  - **RemoteOnly:** In this mode, unhandled errors are displayed differently for browsers on remote computers and browsers on the web server. In a production site, all browsers are remote and errors are displayed on the page that you specify in the **defaultRedirect** attribute. In development situations, the developer often browses to the site on the local computer. When an error occurs, the developer is directed to the default ASP.NET error page, which includes debugging information such as the stack trace. Use the **RemoteOnly** mode on development computers.
- **defaultRedirect:** This attribute sets the name of the view that will be used to display unhandled exceptions. When you create this view, place it in the **Views/Shared** folder.

The code in the following example enables custom errors and specifies that unhandled exceptions are forwarded to the **customError** view.

Configuring Custom Errors

```
<system.web>
      <customErrors mode="On" defaultRedirect="CustomError" />
</system.web>
```

HTTP error pages

Some Hypertext Transfer Protocol-level (HTTP) errors cannot be caught by custom MVC error views as specified in Web.config or the **[HandleError]** annotation. For such errors, you can use the **<error>** element in Web.config, as a child element of the **<customErrors>** element.

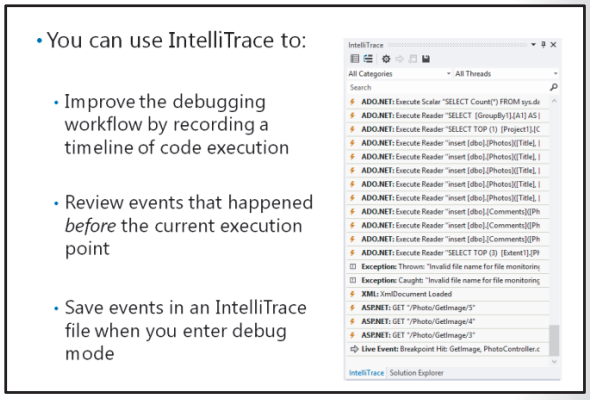
In the following example, the **<error>** element is used for specific HTML files to respond to HTTP 500 errors.

Catching HTTP Error Codes

```
<customErrors mode="On" defaultRedirect="CustomError" >
     <error statusCode="500" redirect="~/Error/Error500.html" />
</customErrors>
```

Question: You have switched off custom errors in the Web.config file. When you run your application without debugging, an exception is thrown in a **try/catch** block. What page displays the exception to the user?

Using Visual Studio IntelliTrace in MVC



When something goes wrong in your web application, whether it happens during development or in production, you need information about the state of the application, to diagnose the problem. Sometimes, a small amount of information is sufficient. For example, you may be able to see from the URL query string that a link was formulated wrongly. Often, however, problems arise in more obscure places and you need as much information as possible to perform a diagnosis. IntelliTrace is a feature of Visual Studio that assists debugging by presenting as much information as possible. IntelliTrace was introduced with Microsoft Visual Studio 2010 Ultimate.

The Traditional Debugging Workflow

When developers debug malfunctioning code, they run the application in Visual Studio debugging mode. If an unhandled exception occurs, Visual Studio breaks execution and displays details such as the exception class and the exception message. You can use the Locals, Call Stack, and Error List windows to determine the values of variables and properties that may have caused the exception. All these pieces of information are presented as they were when the error occurred.

If a problem originated before the error occurred, you must halt debugging, set a break point, and then restart debugging. Again, Visual Studio halts execution, but this time, it is at the break point. You can use the same windows to gain information and step through code, line by line, to see how values change, and diagnose the error.

Two problems can occur in this workflow:

- If you set the break point too late, you will miss the origin of the error. You must set another break point and restart debugging.
- If you are stepping through many lines of code, you can miss the error by selecting Debug|StepInto or pressing the F11 shortcut too many times. This is easy to do when there are hundreds of lines of code to step through.

Debugging With IntelliTrace

IntelliTrace improves this workflow and addresses the two problems that arise during the traditional debugging workflow by recording a timeline of code execution. When you run an application in the debug mode, IntelliTrace records all the events that occur in the background. This can include exceptions, debugging events such as break points, and other events that you can configure.

As before, when the debugger encounters an unhandled exception or a break point, Visual Studio halts execution and displays the details of any exception. However, the developer can now see all the recorded events in the IntelliTrace pane. The current event appears at the end of the list and is named **Live Event.** You can see all the previous events in the list and can click them for more details. In this way, you can investigate what happened before the exception or the break point.

When you click an earlier event in the IntelliTrace pane, the Locals, Call Stack, and Error List windows display the state of the application at the moment of the earlier event. The line of code that corresponds to the selected event is highlighted in orange color. Because of this highlighting, two problems that arise during the traditional debugging workflow are eased, namely, missing the origin of error and stepping too far through the code. Because the relevant code is highlighted, if you miss the origin of the error or step too far through the code, you can get back to earlier events and investigate values.

To help you locate the correct event in the IntelliTrace window, you can filter by event category or execution thread, or use the search option.

IntelliTrace Configuration Options

To configure IntelliTrace, click **DEBUG**, click **IntelliTrace**, and then click **Open IntelliTrace Settings**. You can enable or disable IntelliTrace event recording and specify a location and maximum size for the IntelliTrace files.

Most of the configuration options concern the amount of data to record in IntelliTrace files. The default settings have a mimimal affect on performance, but if those are insufficient to diagnose a problem, you can choose to log more information to target the problem. This may compromise performance, but remember that IntelliTrace information is only recorded during debugging. Your settings will not degrade performance after the application is deployed to a web server. Under **General**, you can set IntelliTrace to record IntelliTrace events only, or both IntelliTrace events and call information, which includes all calls to methods and classes. You can configure what is included in IntelliTrace events by using the **IntelliTrace Events** list. For example, by default, ASP.NET events such as error pages, page post backs, and redirects are included for MVC applications.

Saving and Opening IntelliTrace Files

IntelliTrace automatically saves events in an IntelliTrace file when you enter debug mode. The file is automatically deleted when you close Visual Studio, to ensure that IntelliTrace files do not over-use disk space. Note that this means that the IntelliTrace pane includes events from the current debugging session and any previous debugging sessions run since you started Visual Studio. However, these files are temporary.

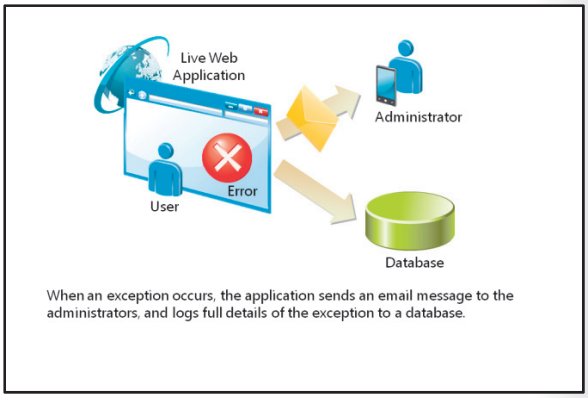
You can avoid losing an IntelliTrace file when you close Visual Studio by saving it for later analysis. To do this, click **DEBUG**, click **IntelliTrace**, and then click **Save IntelliTrace Session**. The file is saved with an .iTrace extension. If you double-click an .iTrace file, Visual Studio displays the threads, exceptions, modules, and other information that the file contains.

Additional Reading: For more information about IntelliTrace and how to use it, see http://go.microsoft.com/fwlink/?LinkID=288962&clcid=0x409

Question: You are looking at IntelliTrace events and suspect that the error originated in a class written by a colleague who works in another country. How can you help the colleague to see the events that you can see in IntelliTrace after you close Visual Studio?

Logging Exceptions

Exceptions that you face during development can be investigated and debugged by using the Visual Studio debugging tools, including IntelliTrace. In an ideal situation, no exceptions would arise when your web application is complete and deployed for users over the Internet. However, in the real world, unforeseen circumstances arise resulting in exceptions. For example, database failures, network issues, and configuration errors in any part of the system can cause exceptions.



You have already seen how to present a branded and informative custom error page to users when

exceptions occur. It is also appropriate to log the exceptions in a production web application, so the administrators and developers can assess the extent of the problem, remove the cause, and improve the robustness of the code.

Writing Error Logging Code

Before you write code to log exceptions in your web application, consider where exceptions should be recorded. Generally, web applications log exceptions to an Extensible Markup Language (XML) file. This approach can be efficient, especially if the text file is kept on a separate drive from the database of the web application and other critical, commonly accessed files. However, the file size of the XML files can grow large, making them difficult to analyze.

A common approach is to log errors to tables in a database. These tables can be a part of a single web application database, or you can place them in a dedicated database away from product catalogs, user profiles, and other information. By logging exceptions to a database, you create a list of exceptions that can be easily searched and analyzed. You can also build a webpage to present these logged exceptions to administrators so that they can access the details from a browser.

You should also consider sending email messages or other types of messages to administrators and developers when exceptions occur. This approach ensures that administrators and developers hear of an error very quickly. It does not require them to review the exceptions logged in the database. Frequently, error handling code in live applications both logs to a database and send email messages to administrators and developers.

Where to Write Error Logging Code

When you decide where to write code that logs errors, you should consider that errors might arise in almost any part of your application. You should choose an approach that enables you to write error logging code once that will run for any exception anywhere in your application.

For example, it is not appropriate to write error logging code in individual **try/catch** blocks. If you do this, you will have to create a **try/catch** block for every procedure in your application, and write duplicate logging code in the **catch** block.

A more effective approach is as follows:

- Create a custom base controller class for your web application. This class inherits from the **System.Web.Mvc.Controller** base class.
- In your custom base controller class, override the **OnException()** method. Place you error logging code in this method.
- When you create controllers, inherit from your custom base controller class, instead of **System.Web.Mvc.Controller**.

In this way, you can write error logging code only once in the overridden **OnExcpetion()** method. This catches exceptions from any of the inheriting controllers.

Using Third-Party Logging Tools

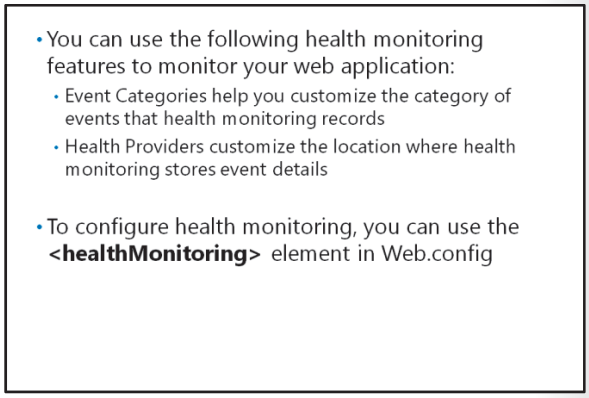
Because exception logging is a very common functional requirement for web applications, there are many third-party solutions that you can choose if you do not want to write your own logging code. Many of these are available within Visual Studio from the NuGet package manager.

Perhaps, the most commonly used and widely supported error logging package is Error Logging Modules and Handlers (ELMAH). ELMAH can be used with any ASP.NET web application and can log exceptions to XML files or a wide variety of databases, including Microsoft SQL Server, Oracle, Microsoft SQL Server Compact, MySQL, and others. In addition, ELMAH includes webpages that you can use to remotely view exception lists and details, and ELMAH can send alert email messages to the address that you configure.

Question: You want to ensure that developers can review exception details that arise during debugging. How should you approach error logging code?

Health Monitoring

Since ASP.NET version 2.0, you can use health monitoring to record what happens in a web application. Health monitoring can record exceptions through error logging techniques. However, health monitoring can also record a wide range of other events such as application starts and stops, failed logon attempts, membership events, and user input validation errors. Some pieces of these information are difficult to access from any other source.



Health Monitoring Features

You can customize the category of events that health monitoring records. Each event is within one of the five following categories.

Event Category	Description
Application Lifetime Events	Application lifetime events include application starts and stops.
Audit Events	Audit events include logons, logoffs, and logon failures.
Error Events	Error events incude .NET Framework exceptions and HTTP errors such as 404 errors.
Request Processing Events	Request processing events include events that occur as ASP.NET controllers, models, and views receive and respond to webpage requests.
Heartbeats	Heartbeats are periodic events that health monitoring can raise and record to confirm that the application is running.

You can also customize the location where health monitoring stores event details by configuring health monitoring providers. You can choose from the following providers.

*Health Monitoring Provi*der	Description
EventLogWebEventProvider	EventLogWebEventProvider sends events to the Windows Event Log. These events appear in the Windows log.
SqlWebEventProvider	SqlWebEventProvider sends events to a SQL Server database.
WmiWebEventProvider	WmiWebEventProvider sends events to Windows Management Infrastructure client.
SimpleMailWebEventProvider	SimpleMailWebEventProvider sends events in an email message sent to an address you specify.
TemplatedMailWebEventProvider	TemplatedMailWebEventProvider also sends events in an email message, but you can use a template email to

*Health Monitoring Prov*ider	Description
	format the information.
TraceWebEventProvider	TraceWebEventProvider sends events to the ASP.NET tracing system.

Configuring Health Monitoring

You configure health monitoring by using the **<healthMonitoring>** element within the **<system.web>** element in Web.config. You can use elements within **<healthMonitoring>** to add and configure providers and define rules that govern how events of different types are recorded.

The following code example shows how to configure health monitoring to record application life cycle events to a SQL Server database.

Configuring Health Monitoring

```
<system.web>
   <connectionStrings>
      <add name="HealthDB" connectionString="your connection string here" />
   </connectionStrings>
   <healthMonitoring enabled="true">
      <providers>
         <add name="sqlProvider"
            type="System.Web.Management.SqlWebEventProvider"
            connectionStringName="HealthDB"
            buffer="false"
            bufferMode="Notification" />
      </providers>
      <rules>
         <add name="LifeCycle"
            provider="sqlProvider"
            eventName="Application Lifetime Events" />
      </rules>
   </healthMonitoring>
<system.web>
```

Additional Reading:

For full details of all the options that you can configure in Web.config for health monitoring, see http://go.microsoft.com/fwlink/?LinkID=293685&clcid=0x409

If you configure health monitoring to use the **SqlWebEventProvider**, ASP.NET will, by default, create a database file called, ASPNETDB.MDF, in the App_Data folder of the web application. You can alter this location by using the connection string in Web.config. However, health monitoring will only work if certain data tables are present in the database that you connect to. You can prepare the database with the correct tables by using the aspnet_regsql.exe tool. This tool is found in the %WINDOWS%\Microsoft.NET\Framework\version directory.

The following command prepares a SQL Server database for health monitoring, authenticating as the current user.

```
aspnet_regsql.exe –E –S HealthDatabaseServer –d HealthDatabase –A w
```

Question: After using your recently-developed web application, several users are facing failed logon attempts. As a result, to track these exceptions, you have decided to set up a monitoring system that will

send an email message to a specified email address. Which health monitoring provider will you use to fix this?

Lab: Testing and Debugging ASP.NET MVC 4 Web Applications

Scenario

The Photo Sharing application is in the early stages of development. However, frequent errors are hindering the productivity of the development team. The senior developer advises that you intercept exceptions and other flaws as early as possible. You have been asked to perform unit tests of the PhotoController to ensure that all scenarios work as expected and to avoid problems later in the web application development life cycle. You have also been asked to ensure that when critical errors occur, developers can obtain helpful technical information.

Objectives

After completing this lab, you will be able to:

- Perform unit tests of the components of an MVC web application.
- Configure an exception handling strategy for an MVC web application.
- Use Visual Studio debugging tools against a web application.

Lab Setup

Estimated Time: 90 minutes

Virtual Machine: **20486B-SEA-DEV11**

User name: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

- On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.
- In the navigation pane of the **Options** dialog box, click **Package Manager**.
- Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Performing Unit Tests

Scenario

In this exercise, you will:

- Create a test project and write the following tests.
  - **Test_Index_Return_View:** This test checks that the Index action returns a view named Index.
  - **Test_PhotoGallery_Model_Type:** This test checks that the _PhotoGallery action passes an enumerable list of **Photo** objects to the _PhotoGallery partial view.
  - **Test_GetImage_Return_Type:** This test checks that the GetImage action returns a file and not a view.
  - **Test_PhotoGallery_No_Parameter:** This test checks that when you call the _PhotoGallery action without any parameters, the action passes all the photos in the context to the _PhotoGallery partial view.

- **Test_PhotoGallery_Int_Parameter:** This test checks that when you call the _PhotoGallery action with an **integer** parameter, the action passes the corresponding number of photos to the _PhotoGallery action.

- Implement a repository.
- Refactor the PhotoController to use a repository.
- Refactor tests to use a mock repository.

 **Note:** The tests you add to the solution in this exercise will improve the quality of code and prevent bugs as development proceeds. However, this exercise does not conform to the principles of TDD because the PhotoController class already exists. In TDD, you would create these and other tests first, and then create a PhotoController class that passes the tests.

The main tasks for this exercise are as follows:

1. Create a test project.
2. Write the tests.
3. Implement a repository.
4. Refactor the photo controller to use the repository.
5. Refactor the tests to use a mock repository.
6. Add further tests.

▶ Task 1: Create a test project.

1. Start the virtual machine and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **PhotoSharingApplication** solution from the following location:
   - File location: **Allfiles (D):\Labfiles\Mod06\Starter\PhotoSharingApplication**
3. Add a new project to the solution for unit tests by using the following information:
   - Template: **Unit Test Project**
   - Name: **PhotoSharingTests**
4. Add a reference to the **PhotoSharingApplication** project in the **PhotoSharingTests** project.
5. Add a reference to the **System.Web.Mvc** assembly in the **PhotoSharingTests** project by using the following information:
   - MVC version: **4.0.0.0**

▶ Task 2: Write the tests.

1. Rename the UnitTest1 class as PhotoControllerTests.
2. Rename the TestMethod1 method as Test_Index_Return_View.
3. Add **using** statements for the following namespaces:
   - **System.Collections.Generic**
   - **System.Web.Mvc**
   - **PhotoSharingApplication.Models**

- **PhotoSharingApplication.Controllers**

4. In the **Test_Index_Return_View** test, create a new **PhotoController**, call the **Index** action, and assert that the name of the result view is **Index**.
5. Add a new test method by using the following information:
   - Annotation: **[TestMethod]**
   - Scope: **public**
   - Return type: **void**
   - Name: **Test_PhotoGallery_Model_Type**
   - Parameters: **None**
6. In the **Test_PhotoGallery_Model_Type** test, create a new **PhotoController**, call the **_PhotoGallery** action, and assert that the type of the result model is **List<Photo>**.
7. Add a new test method by using the following information:
   - Annotation: **[TestMethod]**
   - Scope: **public**
   - Return type: **void**
   - Name: **Test_GetImage_Return_Type**
   - Parameters: None
8. In the **Test_GetImage_Return_Type** test, create a new **PhotoController**, call the **GetImage** action, and assert that the result type is **FileContentResult**.
9. Run all the tests in the **PhotoSharingTests** project and examine the results.

▶ Task 3: Implement a repository.

1. Add a new interface called **IPhotoSharingContext** to the Models folder in the **PhotoSharingApplication** project.
2. Set public scope to the new interface.
3. Add the **Photos** property to the **IPhotoSharingContext** interface by using the following information:
   - Type: **IQueryable<Photo>**
   - Name: **Photos**
   - Access: **Read only**
4. Add the **Comments** property to the **IPhotoSharingContext** interface by using the following information:
   - Type: **IQueryable<Comment>**
   - Name: **Comments**
   - Access: **Read only**
5. Add the **SaveChanges** method to the **IPhotoSharingContext** interface by using the following information:
   - Return type: **Integer**
   - Name: **SaveChanges**
6. Add the **Add** method to the **IPhotoSharingContext** interface by using the following information:

- Return type: **T**, where **T** is any class
- Parameter: an instance of **T** named **entity**

7. Add the **FindPhotoById** method to the **IPhotoSharingContext** interface by using the following information:
   - Return type: **Photo**.
   - Parameter: an integer named **ID**
8. Add the **FindCommentById** method to the **IPhotoSharingContext** interface by using the following information:
   - Return type: **Comment**
   - Parameter: an integer named **ID**
9. Add the **Delete** method to the **IPhotoSharingContext** interface by using the following information:
   - Return type: **T**, where **T** is any class.
   - Parameter: An instance of **T** named **entity**.
10. Ensure that the **PhotoSharingContext** class implements the **IPhotoSharingContent** interface.
11. In the **PhotoSharingContext** class, implement the **Photos** property from the **IPhotoSharingContext** interface and return the **Photos** collection for the **get** method.
12. In the **PhotoSharingContext** class, implement the **Comments** property from the **IPhotoSharingContext** interface and return the **Comments** collection for the **get** method.
13. In the **PhotoSharingContext** class, implement the **SaveChanges** method from the **IPhotoSharingContext** interface and return the results of the **SaveChanges** method.
14. In the **PhotoSharingContext** class, implement the **Add** method from the **IPhotoSharingContext** interface and return a **Set<T>** collection with **entity** added.
15. In the **PhotoSharingContext** class, implement the **FindPhotoById** method from the **IPhotoSharingContext** interface and return the **Photo** object with requested **ID**.
16. In the **PhotoSharingContext** class, implement the **FindCommentById** method from the **IPhotoSharingContext** interface and return the **Comment** object with requested **ID**.
17. In the **PhotoSharingContext** class, implement the **Delete** method from the **IPhotoSharingContext** interface and return a **Set<T>** collection with **entity** removed.
18. Save all the changes and build the project.

▶ Task 4: Refactor the photo controller to use the repository.

1. In the **PhotoController** class, change the declaration of the **context** object so it is an instance of the **IPhotoSharingContext**. Do not instantiate the **context** object.
2. Add a new constructor to the **PhotoController** class. In the controller, instantiate **context** to be a new **PhotoSharingContext** object.
3. Add a second constructor to the **PhotoController** class that accepts an **IPhotoSharingContext** implementation named **Context** as a parameter. In the constructor, instantiate **context** to be the **Context** object.
4. In the PhotoController Display action, replace the call to context.Photos.Find() with a similar call to context.FindPhotoById().

5. In the **PhotoController Create** action for the POST verb, replace the call to **context.Photos.Add()** with a similar call to **context.Add<Photo>**.
6. In the **PhotoController Delete** action, replace the call to **context.Photos.Find()** with a similar call to **context.FindPhotoById()**.
7. In the **PhotoController DeleteConfirmed** action, replace the call to **context.Photos.Find()** with a similar call to **context.FindPhotoById()**.
8. In the PhotoController DeleteConfirmed action, replace the call to context.Photos.Remove() with a similar call to context.Delete<Photo>.
9. In the PhotoController GetImage action, replace the call to context.Photos.Find() with a similar call to context.FindPhotoById()
10. Run the application with debugging to ensure that the changes are consistent.

▶ Task 5: Refactor the tests to use a mock repository.

1. Add a new folder called **Doubles** to the **PhotoSharingTests** project.
2. Add the FakePhotoSharingContext.cs existing file to the Doubles folder from the following location:
   - **Allfiles (D):\Labfiles\Mod06\Fake Repository\ FakePhotoSharingContext.cs**

Note: This class will be used as a test double for the Entity Framework context.

3. In the PhotoControllerTests.cs file**,** add **using** statements for the following namespaces:
   - **System.Linq**
   - **PhotoSharingTests.Doubles**
4. In the **Test_Index_Return_View** method, create a new instance of the **FakePhotoSharingContext** class and pass it to the **PhotoController** constructor.
5. In the **Test_PhotoGallery_Model_Type** method, create a new instance of the **FakePhotoSharingContext** class, add four new **Photo** objects to the class, and then pass them to the **PhotoController** constructor.
6. In the **Test_GetImage_Return_Type** method, create a new instance of the **FakePhotoSharingContext** class.
7. Add four new **Photo** objects to the **context.Photos** collection. Use the following information to add each new **Photo** object:
   - PhotoID: a unique integer value
   - PhotoFile: a new byte array of length 1
   - ImageMimeType: **image/jpeg**
8. Ensure that the new **FakePhotoSharingContext** object is passed to the **PhotoController** constructor.
9. Run all the tests in the **PhotoSharingTests** project and verify the status of all the tests.

▶ Task 6: Add further tests.

1. In PhotoControllerTests.cs, add a new test method by using the following information:
   - Annotation: **[TestMethod]**
   - Scope: **public**
   - Return type: **void**

- Name: **Test_PhotoGallery_No_Parameter**
- Parameters: None

2. In the **Test_PhotoGallery_No_Parameter** method, create a new instance of the **FakePhotoSharingContext** class, add four new **Photo** objects to the class, and then pass them to the **PhotoController** constructor.
3. Call the **_PhotoGallery** action and store the **PartialViewResult** in a variable named **result**.
4. Cast the **result.Model** property as an **IEnumerable<Photo>** collection and then check that the number of **Photos** in the collection is 4, which is the same as the number of photos you added to the fake context.
5. In the PhotoControllerTests.cs code window, copy and paste the entire **Test_PhotoGallery_No_Parameter** method. Rename the pasted test method as **Test_PhotoGallery_Int_Parameter**.
6. In the **Test_Photo_Gallery_Int_Parameter** method, ensure that the call to the **_PhotoGallery** action passes the integer **3**.
7. Assert that the number of **Photo** objects in the **modelPhotos** collection is **3**, which is the integer you passed to the **_PhotoGallery** action.
8. Run all the tests in this **PhotoSharingTests** project and verify the status of all tests.

Results: After completing this exercise, you will be able to add a set of PhotoController tests defined in the PhotoSharingTests project of the Photo Sharing application.

Exercise 2: Optional—Configuring Exception Handling

Scenario

Now that you have developed unit tests for the Photo Sharing application, you need to configure an exception handling strategy for the MVC web application. This would ensure that when exceptions occur in the development phase of the PhotoSharingApplication project, the controller, action, and exception messages are displayed in a custom MVC error view. You also need to implement a placeholder action for the SlideShow action in the PhotoController view. This action will be completed during a later iteration of the project.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Edit Web.config for exception handling.
2. Create a custom error view.
3. Configure errors in the PhotoController class.
4. Raise errors.

▶ Task 1: Edit Web.config for exception handling.

1. Open the Web.config file in the root level folder of the **PhotoSharingApplication** project.
2. Add the **<customErrors>** element to the **<system.web>** element by using the following information:
   - Parent element: **<system.web>**
   - Element: **<customErrors>**

- Mode: **On**
- defaultRedirect: **ErrorPage**

3. Add the **<error>** element to the **<customErrors>** element by using the following information:
   - Parent element: **<customErrors>**
   - Element: **<error>**
   - statusCode: **500**
   - redirect: **Error.html**
4. Add a new HTML page to the **PhotoSharingApplication** project by using the following information:
   - Template: **HTML Page**
   - Name: **Error.html**
5. In the **Error.html** file, set the contents of the **TITLE** element to **Error**.
6. Add content to the **Error.html** file to explain the error to users.

▶ Task 2: Create a custom error view.

1. Add a new view to the **Shared** folder by using the following information:
   - Name of the view: **Error**
   - View type: **Not strongly typed**
   - Layout or master page: **None**
2. In the **Error.cshtml** file, set the content of the **TITLE** element to **Custom Error**.
3. Set the **@model** for the **Error.cshtml** to **System.Web.Mvc.HandleErrorInfo**.
4. In the **DIV** element, render an **H1** element by using the following information:
5. Content: MVC Error
6. In the DIV element, render the ControllerName property of the Model object.
7. In the DIV element, render the ActionName property of the Model object.
8. In the **DIV** element, render the **Exception.Message** property of the **Model** object.
9. Save all the changes made to the **Error.cshtml** file.

▶ Task 3: Configure errors in the PhotoController class.

1. Modify the **PhotoController** class to send errors to the **Error.cshtml** view by using the following information:
   - Class: **PhotoController**
   - Annotation: **HandleError**
   - View: **Error**
2. Add a new action to the **PhotoController** class by using the following information:
   - Scope: **public**
   - Return type: **ActionResult**
   - Name: **SlideShow**
   - Parameters: **None**

3. In the new action, throw an exception by using the following information:
   - Type: **NotImplmentedException**
   - Message: **The SlideShow action is not yet ready**

▶ Task 4: Raise errors.

4. Start debugging and display **Sample Photo 5**.
1. In the Internet Explorer window, request the relative URL and view the error details.
   - URL: **/Photo/Display/malformedID**
2. In the Internet Explorer window, request the relative URL.
   - URL: **/Photo/SlideShow**
3. Use the IntelliTrace pane to investigate the exception.
4. Stop debugging and close Visual Studio.

Results: After completing this exercise, you will be able to:

Configure a custom error handling strategy for an MVC application.

Question: When you ran the tests for the first time in Exercise 1, why did **Test_Index_Return_View** pass, while **Test_GetImage_Return_Type** and **Test_PhotoGallery_Model_Type** failed?

Question: In Exercise 1, why did all the tests pass during the second run?

Module Review and Takeaways

In this module, you became familiar with various techniques that you can use to eliminate bugs from your ASP.NET MVC 4 web application. This begins early in the development phase of the project, when you define unit tests that ensure that each method and class in your application behaves precisely as designed. Unit tests are automatically rerun as you build the application so you can be sure that methods and classes that did work are not broken by later coding errors. Exceptions arise in even the most well-tested applications because of unforeseen circumstances. You also saw how to ensure that exceptions are handled smoothly, and how error logs are optionally stored for later analysis.

Best Practice: If you are using TDD or Extreme Programming, define each test before you write the code that implements a requirement. Use the test as a full specification that your code must satisfy. This requires a full understanding of the design.

Best Practice: Investigate and choose a mocking framework to help you create test double objects for use in unit tests. Though it may take time to select the best framework and to learn how to code mock objects, your time investment will be worth it over the life of the project.

Best Practice: Do not be tempted to skip unit tests when under time pressure. Doing so can introduce bugs and errors into your system and result in more time being spent debugging.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
No information appears in the IntelliTrace window.	

Review Question(s)

Question: You want to ensure that the PhotoController object passes a single Photo object to the Display view, when a user calls the **Search** action for an existing photo title. What unit tests should you create to check this functionality?

Tools

NinJect, StructureMap. These are Inversion of Control (IoC) containers, also known as Dependency Injection (DI) frameworks. They create non-test implementations of interfaces in your web application.

Moq, RhinoMocks, NSubstitute. These are mocking frameworks. They automate the creation of test doubles for unit tests.

IntelliTrace. This is a part of Visual Studio that displays application state at the point of an exception or break, and at earlier times.

Health Monitoring. This part of ASP.NET can store health events in a database, log, or other locations for later analysis.

ELMAH. This exception logging tool can store exceptions in database tables, XML files, and elsewhere, and enable administrators to view exception details on a webpage.

Module 07

Structuring ASP.NET MVC 4 Web Applications

Contents:

Module Overview

An MVC web application can grow large and complex and the application can contain many types of information that relate to each other in complicated ways. A key challenge for such web applications is to ensure that users can easily locate the information that interests them. Although a search tool is helpful, you should design your web application in such a way that users can navigate to the page they need through a short sequence of links. At each level, users must know what link to click next. The URLs that users see in the Address bar must be readable and understandable. You need to understand how to present the model classes you have designed in a way that is logical to the user. You also need to ensure that Internet search engines can crawl and analyze your web application structure, so that your application content appears close to the top of the search engine results. You can achieve these goals by learning about ASP.NET routing and navigation.

Objectives

After completing this module, you will be able to:

- Plan suitable information architecture for a given set of functional requirements.
- Add routes to the ASP.NET routing engine and ensure that URLs are user-friendly throughout an MVC web application.
- Add navigation controls to an application to enable users to locate data rapidly.

Lesson 1
Analyzing Information Architecture

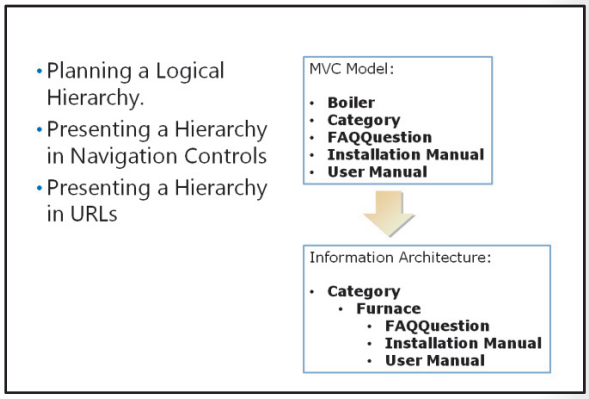
When you analyze use cases and user stories to plan a model for your web application, you must determine the kinds of objects that your web application will manage. For example, if your web application provides technical documentation for a product, you can plan model classes such as installation guides, user guides, frequently asked questions (FAQs), FAQ answers, categories, comments, and other model classes. The model imposes a logical structure of objects on the web application. However, you must also consider how users expect information to be structured. Users may expect a hierarchical structure, in which FAQs, FAQ answers, and comments are presented under the product they relate to. When you create a complex MVC web application, you need to know how to present model classes in a logical manner, so that your web application is user-friendly.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe information architecture.
- Describe search engine optimization and its importance for web developers.

What Is Information Architecture?



When the information your web application manages is complex and multi-faceted, it is easy to present objects in a confusing way. Unless you think carefully about the way users expect information to be structured and how they expect to navigate to useful content, you may unintentionally create an unusable web application. During development, when you work with a limited amount of data, this confusion may not become apparent. Then, when real-world data is added to your database at production deployment time, it becomes clear that the web application is confusing. You can avoid this eventuality by planning the information architecture.

Information architecture is a logical structure for the objects your web application manages. You should design such architecture in a way that users can find content quickly without understanding any technical aspects of your web application. For example, users should be able to locate a technical answer about their product without understanding the database structure or the classes that constitute the model.

Example Scenario: A Technical Documentation Website

To understand the need for information architecture, consider a web application that presents technical information about a company's products for customers and engineers. In this example, the company manufactures domestic heating furnaces, which are installed in homes by qualified heating engineers. The web application is intended to enable customers to locate instructions, hints, and tips. This web application is also intended to enable engineers to obtain the technical documentation on installing and servicing furnaces.

The development team has identified the following simple user stories:

- Customers have a certain problem with their boilers. They want to find a specific FAQ answer that solves the problem. They know the boiler product name and the fuel, but not the product

number. They visit the web application and navigate to the boiler name. They click the FAQ for their boiler name and locate the answer that they need.

- Engineers need the latest installation manual for a boiler. They know the boiler product number, product name, and fuel type. They visit the site and navigate to the boiler name. They click the Manuals link and locate the installation manual.
- Engineers have web applications, which they want to link to a specific boiler name. You want to ensure that the URL is simple and readable for customers and engineers.

You have already planned the following model classes:

- *Furnace*. Each furnace object is a product manufactured and sold by the company.
- *Category*. Categories organize the furnaces by type. For example, you can have categories such as oil-fired, gas-fired, and solid fuel.
- *FAQ*. Each FAQ relates to a single furnace. Each furnace can have many questions. The class includes both Question and Answer properties.
- *Installation Manual*. Each furnace has a single installation manual in the form of a PDF document.
- *User Manual*. Each furnace has a single user manual in the form of a PDF document.

Planning a Logical Hierarchy

You can see from the user stories that FAQ and manuals are both accessed by navigating to the relevant product first. You can also see that the company has different products, and both customers and engineers know the fuel type for a particular furnace. Therefore, you can organize furnaces in categories by fuel type. The following list shows a logical hierarchy of objects, which helps both the customers and engineers find the information they need by clicking through each level:

- Category
  - Furnace
  - FAQ
  - User Manual
  - Installation Manual

Presenting a Hierarchy in Navigation Controls

The information architecture you design should be presented on webpages in the form of navigation controls. Common approaches to such controls include:

- *Site Menus*. Most websites have a main menu that presents the main areas of content. For simple web applications, the main menu may include a small number of static links. For larger web applications, when users click a site menu link, a submenu appears.
- *Tree Views*. A tree view is a menu that shows several levels of information hierarchy. Usually, users can expand or collapse objects at each level, to locate the content they require. Tree views are useful for presenting complex hierarchies in navigable structures.
- *Breadcrumb Trails*. A breadcrumb trail is a navigation control that shows the user where they are in the web application. Usually a breadcrumb trail shows the current pages and all the parent pages in the hierarchy, with the home page as the top level. Breadcrumb trails enable you to understand how a page fits in with the information architecture shown in menus and tree views.

The types of navigation controls you build in your web application depend on how you expect users to find information.

Presenting a Hierarchy in URLs

You can increase the usability of your web application by reflecting the information architecture in the URLs, which the users see in the Address bar of the web browser. In many web applications, URLs often include long and inscrutable information such as Globally Unique Identifiers (GUIDs) and long query strings with many parameters. Such URLs prevent users from manually formulating the address to an item, and these URLs are difficult to link to a page on your web application. Instead, URLs can be plain and comprehensible, to help users browse through your content.

In MVC web applications, the default configuration is simple, but it is based on controllers, actions, and parameters. The following are some example URLs that follow this default pattern:

- *http://site/Furnace/Details/23:* This URL links to the Furnace controller and the Details action, and it displays the furnace with the ID 23.
- *http://site/FAQQuestion/Details/234:* This URL links to the FAQQuestion controller and the Details action, and it displays the FAQ with the ID 234.
- *http://site/InstallationManual/Details/3654:* This URL links to the InstallationManual controller and the Details action, and it displays the manual with the ID 3654.

As you can see, the web application user is required to understand how controllers, actions, and action parameters work, to formulate URLs themselves. Instead, users can use URLs that are easier to understand, such as the following, because they reflect the information hierarchy:

- *http://site/OilFired/HotBurner2000:* This URL links to a Furnace by specifying the fuel category and the product name. Customers and engineers can understand these values.
- *http://site/OilFired/HotBurner2000/HowToRestartMyFurnace:* This URL links to an FAQ by specifying the furnace name the question relates to.
- *http://site/OilFired/HotBurner2000/:* This URL links to the Installation Manual by specifying the furnace name the manual relates to.

As you can see, these URLs are easy for customers and engineers to understand, because the URLs are based on a logical hierarchy and the information that the users already have. You can control the URLs that your ASP.NET web application uses, by configuring the ASP.NET routing engine.

Question: Why may it be difficult for users to understand URLs based on controllers, actions, and parameters?

What Is Search Engine Optimization?

Most users find web applications by using search engines. Users tend to visit the links that appear at the top of search engine results more frequently than those lower down and those on the second page of results. For this reason, website administrators and developers try to ensure their web application appears high in search engine results, by using a process known as Search Engine Optimization (SEO). SEO ensures that more people visit your web application.

- Use meaningful <title> elements
- Use accurate <meta name="keyword"> tags
- Use accurate <meta name="description"> tags
- Use different <title> <meta> elements on each page
- Choose a domain name that includes keywords
- Use keywords in heading elements
- Ensure that navigation controls enable web bots to crawl your entire web application
- Ensure that URLs do not include GUIDs or long query text

Search engines examine the content of your web application, by crawling it with a program called a

web bot. If you understand the priorities that web bots and search engine indexes use to order search results, you can create a web application that conforms to those priorities and thereby appears high in search engine results.

SEO Best Practices

Various search engines have different web bots with different algorithms to prioritize results. The complete details of these algorithms are not usually published. However, if you adopt the following best practices, your site has a good chance of appearing high in search results:

- Ensure that each webpage in your web application has a meaningful <title> element in the <head> section of the HTML.
- Ensure that you include a <meta name="keywords"> tag in the <head> element of each page. The content attribute of this element should include keywords that describe the content of the page accurately.
- Ensure that you include a <meta name="description"> tag in the <head> element of each page. The content attribute of this element should include a sentence that describes the content of the page accurately.
- Ensure that the <title> element and the <meta> elements are different for each page in your web application.
- Choose a domain name that includes one or more of your most important keywords.
- Ensure that keywords appear in the <h1>, <h2>, or <h3> elements of your webpage.
- Ensure that navigation controls enable web bots to crawl your entire web application. For example, you may have content in your site that users can only find with the search tool, not by clicking through links. As web bots cannot use search tools, this content will not be indexed.
- Ensure that URLs do not include GUIDs or long query text.

SEO and Web Application Structure

Information architecture is a subject that is closely related to SEO. This is because both information architecture and SEO are relevant to the structure, hierarchy, and accessibility of the objects in your web application. Users click links on menus to navigate to the pages that interest them. Web bots use the same links to navigate the web application and crawl its content. Users prefer URLs without GUIDs and long query text because they are meaningful. Web bots often ignore links with GUIDs and long query text in them. In addition, when keywords appear in URLs, web bots prioritize a webpage in search results.

As an ASP.NET MVC developer, you must understand SEO principles and use them whenever you write code, to ensure that you do not damage the search engine positioning of your web application. Views are critical to SEO because they render <meta> tags, and <title> elements. Routes and the configuration of the routing engine are also critical, because, by using routes, you can control the URLs that your web application generates.

Question: A developer creates a partial view that contains <meta> tags and <title> tags. The developer uses this partial view on every page in the web application. Do these actions conform to SEO best practices?

Lesson 2
Configuring Routes

ASP.NET enables developers to control the URLs that a web application uses, to link the URLs and the content by configuring routes. A route is an object that parses a requested URL, and it determines the controller and action to which the request must be forwarded. Such routes are called incoming routes. HTML helpers also use routes when they formulate links to controllers and actions. Such routes are called outgoing routes. You need to know how to write code that adds a new route to your application. You also need to know how the routing engine interprets a route so that requests go to the appropriate controllers, and users see meaningful URLs.

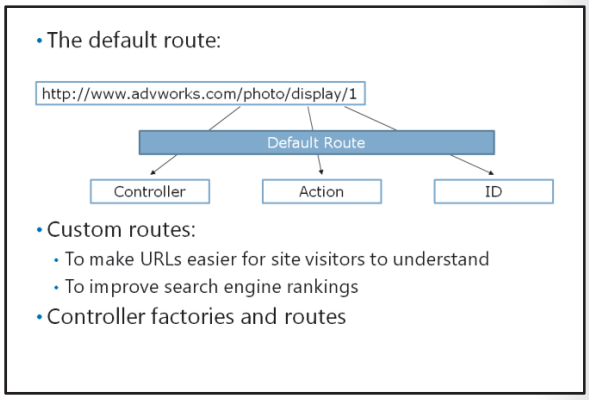
Lesson Objectives

After completing this module, you will be able to:

- Describe the features of the ASP.NET routing engine.
- Describe the benefits of adding routes to an ASP.NET MVC web application.
- Explain how to add and configure routes.
- Explain how to use routes to pass parameters.
- Add routes to manage URL formulation.

The ASP.NET Routing Engine

Routing governs the way URLs are formulated and how they correspond to controllers and actions. Routing does not operate on the protocol, server, domain, or port number of a URL, but only on the directories and file name in the relative URL. For example, in the URL, http://www.advworks.com/photo/display/1, routing operates on the relative path /photo/display/1.



In ASP.NET, routes are used for two purposes:

- To parse the URLs requested by browsers. This analysis ensures that requests are forwarded to the right controllers and actions. These are called incoming URLs.
- To formulate URLs in webpage links and other elements. When you use helpers such as **Html.ActionLink()** and **Url.Action()** in MVC views, the helpers construct URLs according to the routes in the routing table. These are called outgoing URLs.

When you create an MVC web application in Visual Studio by using a project template, the application has a default route. You must understand this default route before you consider adding extra routes to manipulate URLs in the application.

Note: You can use routes and the routing engine to govern relative URLs in any ASP.NET 4.5 application and not just in those that use the MVC programming model.

The Default Route

The default route is simple but logical and works well in many web applications. The default route examines the first three segments of the URL. Each segment is delimited by a forward slash:

- The first segment is interpreted as the name of the controller. The routing engine forwards the request to this controller. If a first segment is not specified, the default route forwards the request to a controller called **Home**.
- The second segment is interpreted as the name of the action. The routing engine forwards the request to this action. If a second segment is not specified, the default route forwards the request to a controller called **Index**.
- The third segment is interpreted as an ID value, which is passed to the action as a parameter. The parameter is optional, so if a third segment is not specified, no default value is passed to the action.

You can see that when the user requests the URL, http://www.advworks.com/photo/display/1, the default route passes the ID parameter **1** to the **Display** action of the **Photo** controller.

The following code shows how the default route is implemented in new ASP.NET MVC 4 applications.

The Default Route

```
routes.MapRoute(
   name: "Default",
   url: "{controller}/{action}/{id}",
   defaults: new {
      controller = "Home",
      action = "Index",
      id = UrlParameter.Optional
   }
);
```

Custom Routes

Developers add their own custom routes to a web application for two main reasons:

- *To make URLs easier for site visitors to understand.* A URL such as http://www.advworks.com/photo/display/1 is logical to the visitor, but it requires some knowledge to formulate such URLs. In this case, to type the right URL in the Address bar of the web browser, the user must know some information. This information includes the controller name, **Photo**, the action name, **Display**, and the **ID** of the photo of interest. If you use Globally Unique Identifiers (GUIDs) in the database, the **ID** segment of the URL can be long and difficult to remember. Ideally, you should consider what users know and create routes that accept that information. In this example, users may know the title of a photo that they had seen earlier in your application. You should create a route that can interpret a URL such as http://www.advworks.com/photo/title/My%20Photo%20Title, and forward the URL to an appropriate action to display the right photo. Although users usually click links to make requests, friendly URLs like these make a site easier to use and link to from other sites.
- *To improve search engine rankings.* Search engines do not prioritize webpages that have GUIDs or long query text in the URL. Some web bots do not even crawl such links. In addition, some search engines boost a page's ranking when one or more of its keywords appear in the URL. User-friendly URLs are therefore a critical tool in SEO.

Controller Factories and Routes

MVC uses controller factories that help create an instance of a controller class to handle a request. MVC uses action invokers to call the right action and pass parameters to that action method. Both controller

factories and action invokers refer to the routing table to complete their tasks. The following are steps that MVC conducts when a request is received from a web browser:

1. An **MvcHandler** object creates a controller factory. The controller factory is the object that instantiates a controller to respond to the request.
2. The controller factory consults the routing table to determine the right **Controller** class to use.
3. The controller factory creates a **Controller** object, and the **MvcHandler** calls the **Execute** method in that controller.
4. The **ControllerActionInvoker** examines the request URL and consults the routing table to determine the action in the **Controller** object to call.
5. The **ControllerActionInvoker** uses a model binder to determine the values that should be passed to the action as parameters. The model binder consults the routing table to determine if any segments of the URL should be passed as parameters. The model binder can also pass parameters from a posted form, from the URL query text, or from uploaded files.
6. The **ControllerActionInvoker** runs the action. Often, the action creates a new instance of a model class, perhaps by querying the database with the parameters that the invoker passed to it. This model object is passed to a view, to display results to the user.

As you can see, you can use routes to manage the behavior of controller factories, action invokers, and model binders, because all these objects refer to the routing table. MVC is highly extensible; therefore, developers can create custom implementations of controller factories, action invokers, and model binders. However, by using routes, you can usually implement the URL functionality that you need with the default implementations of these classes. You should ensure that routes cannot implement your required functionality before you plan to customize controller factories, action invokers, or model binders.

Question: A user wants to edit a comment that the user created in your MVC application. You have not created any custom routes. What URL do you think the user must request, to see the edit view with the right comment?

Adding and Configuring Routes

Every MVC web application has a **RouteTable** object in which routes are stored, in the **Routes** properties. You can add routes to the **Routes** property by calling the **MapRoute()** method.

- Understand the properties of a route:
  - Includes Name, URL, Constraints and Defaults
- Analyze the default route code:
  - Specifies **Name**, **URL** ,and **Defaults** properties
- Create Custom Routes:
  - Involves calling the **routes.MapHttpRoute()** method
- Understand the precedence of routes:
  - Add routes to the **RouteTable.Routes** collection in the appropriate order

```
routes.MapRoute(
  name: "Default",
  url: "{controller}/{action}/{id}",
  defaults: new {
    controller = "Home",
    action = "Index",
    id = UrlParameter.Optional }
);

routes.MapRoute(
  name: "PhotoRoute",
  url: "photo/{id}",
  defaults: new {
    controller = "Photo",
    action = "Details" },
  constraints: new {
    id = "[0-9]+" }
);
```

In the Visual Studio project templates for MVC 4, a dedicated **RouteConfig.cs** code file exists in the **App_Start** folder. This file includes the **RouteConfig.RegisterRoutes()** static method where the default route is added to the **RouteTable** object. You can add custom routes in this method. The **Global.asax.cs** file includes a call to **RouteConfig.RegisterRoutes()** in the **Application_Start()** method, which means that routes are added to the routing table whenever the MVC application starts.

Properties of a Route

Before you can add a route, you must understand the properties of a route. This is to ensure that these properties match the URL segments correctly and pass requests and parameters to the right location. The properties of a route include the following:

- *Name*. This string property assigns a name to a route. It is not involved in matching or request forwarding.
- *URL*. This string property is a URL pattern that is compared to a request, to determine if the route should be used. You can use segment variables to match a part of the URL. You can specify a segment variable by using braces. For example, if you specify the URL, "photo/{title}", the route matches any request where the relative URL starts with the string, "photo/", and includes one more segment. The segment variable is "title" and can be used elsewhere in the route.
- *Constraints*. Sometimes you must place extra constraints on the route to ensure that it matches only with the appropriate requests. For example, if you want relative URLs in the form, "photo/34", to specify a photo with the ID "34", you must use a URL property like "photo/{id}". However, observe that this URL pattern also matches the relative URL, "photo/create", because it has one extra segment. For IDs, you must constrain the URL to match only segments comprising digits. You can do this by adding a constraint to the route. The **Constraints** property enables you to specify a regular expression for each segment variable. The route will match a request only if all the segment variables match the regular expressions that you specify.
- *Defaults*. This string property can assign default values to the segment variables in the URL pattern. Default values are used for segment variables when the request does not specify them.

The Default Route Code

The default route specifies the **Name**, **URL**, and **Defaults** properties to obtain controller and action names, and ID values, from the requested relative URL. By examining these properties, you can understand how to construct your own routes.

The following code shows the default route in the Visual Studio MVC project templates.

The Default Route in RouteConfig.cs

```
routes.MapRoute(
   name: "Default",
   url: "{controller}/{action}/{id}",
   defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Observe that the URL pattern specifies three segments: {controller}, {action}, and {id}. This means that a relative URL in the form, "photo/display/23", will be sent to the **Display** action of the **PhotoController** class. The third segment will be passed as a parameter to the action method. Therefore, if the **Display** action accepts a parameter named **id**, the value, **23**, will be passed by MVC.

In the **defaults** property, the {id} segment is marked as optional. This means that the route still matches a relative URL, even if no third segment is specified. For example, the route matches the relative URL, "photo/create", and passes the URL to the **Create** action of the **PhotoController** class.

The **defaults** property specifies a default value of "Index" for the {action} segment. This means that the route still matches a relative URL, even if no second segment is specified. For example, the route matches the relative URL, "photo", and passes the URL to the **Index** action of the **PhotoController** class.

Finally, the **defaults** property specifies a default value of "Home" for the {controller} segment. This means that the route still matches a relative URL, even if no segments are specified, that is, if there is no relative URL. For example, the absolute URL, "http://www.advworks.com", matches this route, and the URL is passed to the **Index** action of the **HomeController** class.

As you can see, with the default route in place, developers build the web application's home page by creating a controller named, **HomeController**, with an action named, **Index**. This action usually returns a view called, **Index**, from the **Views/Home** folder.

Creating Custom Routes

You can add custom routes by calling the **routes.MapHttpRoute()** method, just like the default route.

In the following code example, a constraint is used to ensure that only digits match the {id} segment variable.

A Custom Route

```
routes.MapRoute(
   name: "PhotoRoute",
   url: "photo/{id}",
   defaults: new { controller = "Photo", action = "Details" },
   constraints: new { id = "[0-9]+" }
);
```

Observe that the URL pattern in the route matches any relative URL that starts with "photo/" and has one or more segments. However, in the constraints property, the route specifies a regular expression for the {id} segment variable. This regular expression matches only if the segment is composed of digits that range from 1 through 9. If the user requests the relative URL, "photo/234", then this route matches, and the routing engine forwards the request to the **Details** action of the **PhotoController** class. If the user requests the relative URL, "photo/create", then this route does not match and a subsequent route must be used to forward the request.

Precedence of Routes

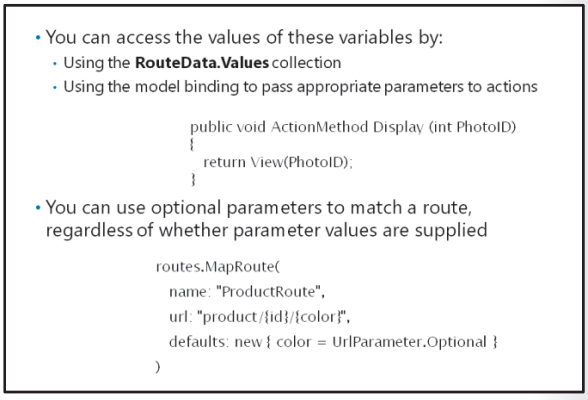
Routes are evaluated by the routing engine in the order with which they are added to the **RouteTable.Routes** collection. If a route matches, the routing engine uses it and does not evaluate any later route in the collection. If a route does not match, the routing engine ignores it and evaluates the next route in the collection. For this reason, you must add routes to the **RouteTable.Routes** collection in the appropriate order. You can add the most specific routes first, such as routes that have no segment variables and no default values. Routes with constraints should be added before routes without constraints.

The route named "Default" matches any request, including those with no relative URL. This default route should be the last that you add to the **RouteTable.Routes** collection. If the routing engine evaluates this route, the route is always used. Any routes added after the "Default" route are never used.

Question: A developer has removed all code from the **Application_Start()** method in **Global.asax.cs**. When the developer runs the application, he or she receives 404 errors for any request, regardless of the relative URL. Why does this occur?

Using Routes to Pass Parameters

The routing engine separates the relative URL in a request into one or more segments. Each forward slash delimits one segment from the next. If you want one of the segments to specify the controller name, you can use the {controller} segment variable. The controller factory always interprets this variable as the controller to instantiate. Alternatively, to fix a route to a single controller, you can specify a value for the **controller** variable in the **defaults** property. In a similar manner, if you want one of the segments to specify the action method, you can use the {action} segment



variable. The action invoker always interprets this variable as the action to call. Alternatively, to fix a route to a single action, you can specify a value for the **action** variable in the **defaults** property.

Segment variables or default values with other names have no special meaning to MVC and are passed to actions. You can access the values of these variables by using one of two methods: the **RouteData.Values** collection or by using model binding to pass values to action parameters.

The RouteData.Values Collection

In the action method, you can access the values of any segment variable by using the **RouteData.Values** collection. Consider the following example:

- In **RouteConfig.cs**, you have defined a route with the URL pattern "{controller}/{action}/{title}".
- The user requests the relative URL, "photo/edit/my%20photo".

The **RouteData.Values** collection contains the following values, which you can use in the action method:

- **RouteData.Value["controller"]**: This value has the value, "photo".
- **RouteData.Value["action"]**: This value has the value, "edit".
- **RouteData.Value["title"]**: This value has the value, "my%20photo".

Using Model Binding to Obtain Parameters

Consider the following action method.

```
public void ActionMethod Display (int PhotoID)
{
   return View(PhotoID);
}
```

The default MVC action invoker is a class named, **ControllerActionInvoker**. This invoker uses a class named, **DefaultModelBinder**, to pass the appropriate parameters to actions. This model binder uses the following logic to pass parameters:

1. The binder examines the definition of the action to which it must pass parameters. In the example, the binder determines that the action requires an integer parameter called, **PhotoID**.
2. The binder searches for values in the request, which can be passed as parameters by name. In the example, the binder searches for integers because the action requires integers. The binder searches for values in the following locations in sequential order:
   a. *Form Values*. If the user fills out a form and clicks the Submit button, the binder can find parameters in the **Request.Form** collection.
   b. *Route Values*. If the matched route includes the segment variable {photoid}, the binder can find the value for the **PhotoID** parameter in the **RouteData.Values** collection. This match is case-insensitive.
   c. *Query Strings*. If the user request includes named parameters after a question mark, the binder can find these parameters in the **Request.QueryString** collection.
   d. *Files:* If the user request includes uploaded files, these files can be used as parameters in an action. File parameters in actions must use the **HttpPostedFileBase** type.

Therefore, you can use the default model binder to pass parameters to action strings. If the name of an action method parameter matches the name of a route segment variable, the model binder passes the parameter automatically.

Optional Parameters

Sometimes, you want requests to match a route, regardless of whether parameter values are supplied. For example, you might want to enable users to specify a color for a product.

In the following example, the route added includes an optional color parameter.

A Route with an Optional Parameter

```
routes.MapRoute(
   name: "ProductRoute",
   url: "product/{id}/{color}",
   defaults: new { color = UrlParameter.Optional }
);
```

The default values specify that {color} is optional. If the relative URL has three segments, the route matches, and the third segment is passed to the **RouteData.Values["color"]** value. If the relative URL has only two segments, the route matches, because the {color} is optional. In this case, the **RouteData.Values["color"]** value does not exist. If the relative URL has only one segment, the route does not match.

Question: A developer has replaced the default model binder with a custom model binder. Now, several action methods are throwing exceptions on lines that use action parameters. How can you fix this bug?

Demonstration: How to Add Routes

In this demonstration, you will see how to create URLs by adding routes to the MVC web application. You will also see how to enable visitors to the Operas web application to browse the operas by title.

Demonstration Steps

1. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.
2. On the Operas I Have Seen page, click the **operas I've seen** link.
3. On the Index page, click the **Details** link corresponding to **Cosi Fan Tutte**.
4. In the Address bar of the Windows Internet Explorer window, note that the URL is **http://localhost:**<*portnumber*>**/Opera/Details/1**.

Note: This URL indicates that the controller is **Opera**, the action is **Details**, and the ID is **1**.

5. In the Windows Internet Explorer window, click the **Close** button.
6. In the Solution Explorer pane, expand **OperasWebSite**, expand **Controllers**, and then click **OperaController.cs**.
7. In the OperaController.cs code window, place the mouse cursor at the end of the **Details** action code block, press Enter twice, and then type the following code.

```
public ActionResult DetailsByTitle(string title)
{
}
```

8. In the **DetailsByTitle** action code block, type the following code, and then press Enter.

```
Opera opera = (Opera)(from o in contextDB.Operas
   where o.Title == title
```

```
   select o).FirstOrDefault();
```

9. In the **DetailsByTitle** action code block, after the code that you just entered, type the following code.

```
if (opera == null)
{
   return HttpNotFound();
}
return View("Details", opera);
```

10. In the Solution Explorer pane, under OperasWebSite, expand **App_Start**, and then click **RouteConfig.cs**.
11. In the RouteConfig.cs code window, locate the following code.

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

12. Place the mouse cursor at the end of the call to the **IgnoreRoute()** method, press Enter twice, and then type the following code.

```
routes.MapRoute(
   name: "OperaTitleRoute",
   url: "opera/title/{title}",
   defaults: new { controller = "Opera", action = "DetailsByTitle" }
);
```

13. On the **FILE** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Save All**.
14. On the **DEBUG** menu of **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.
15. On the Operas I Have Seen page, click the **operas I've seen** link.
16. In the Address bar of the Windows Internet Explorer window, append the existing URL with **/title/rigoletto**, and then click the **Go** button.

Note: The details of the **Rigoletto** opera are displayed.

17. In the Windows Internet Explorer window, click the **Close** button.
18. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Unit Tests and Routes

Unit tests are small, isolated methods that check a specific aspect of your code. As your project grows in complexity, you can rerun all unit tests to ensure that the verified functionality remains in a working state. If your team uses Test Driven Development (TDD), you can define tests to specify a functional requirement and then write code that passes your test. Such approaches significantly increase the quality of your code, by highlighting bugs as soon as they arise.

A Unit Test for the Routing Table:

```
[TestMethod]
public void Test_Default_Route_ControllerOnly()
{
   //Arrange
   var context = new FakeHttpContextForRouting(
      requestUrl: "~/ControllerName");
   var routes = new RouteCollection();
   MyMVCApplication.RouteConfig.RegisterRoutes(routes);

   // Act
   RouteData routeData = routes.GetRouteData(context);

   // Assert
   Assert.AreEqual("ControllerName", routeData.Values["controller"]);
   Assert.AreEqual("Index", routeData.Values["action"]);
   Assert.AreEqual(UrlParameter.Optional, routeData.Values["id"]);
}
```

You can use unit tests to ensure that routes in your MVC web application behave as designed. If

you do such tests, you can prevent many 404 HTTP errors and internal server errors from reaching web application users. Users generally have a low tolerance for such errors, and if errors arise too often, they will browse to the websites of your competitors.

Creating a HTTP Context Test Double

In a webpage request, ASP.NET stores details of the request in an HTTP Context object. These details include the URL that was requested, details of the browser, and other information. To write a unit test for routes, you must simulate the HTTP Context object by creating a test double class.

The following code shows how to create a fake HTTP context object to use as a test double.

An HTTP Context Test Double

```
    public class FakeHttpContextForRouting : HttpContextBase
    {
        FakeHttpRequestForRouting _request;
        FakeHttpResponseForRouting _response;
        //The constructor enables you to specify a request URL and an Application path in
the test
        public FakeHttpContextForRouting(string appPath = "/", string requestUrl = "~/")
        {
            //Create new fake request and response objects
            _request = new FakeHttpRequestForRouting(appPath, requestUrl);
            _response = new FakeHttpResponseForRouting();
        }
        //This property returns the fake request
        public override HttpRequestBase Request
        {
            get { return _request; }
        }
        //This property returns the fake response
        public override HttpResponseBase Response
        {
            get { return _response; }
        }
    }
    //This is the fake request class
    public class FakeHttpRequestForRouting : HttpRequestBase
    {
        string _appPath;
        string _requestUrl;
        public FakeHttpRequestForRouting(string appPath, string requestUrl)
        {
            _appPath = appPath;
            _requestUrl = requestUrl;
        }
        public override string ApplicationPath
        {
            get { return _appPath; }
        }
        public override string AppRelativeCurrentExecutionFilePath
        {
            get { return _requestUrl; }
        }
        public override string PathInfo
        {
            get { return ""; }
        }
        public override NameValueCollection ServerVariables
        {
            get { return new NameValueCollection(); }
        }
    }
    //This is the fake response class
    public class FakeHttpResponseForRouting : HttpResponseBase
    {
```

```
        public override string ApplyAppPathModifier(string virtualPath)
        {
            return virtualPath;
        }
    }
```

Writing Routing Tests

After creating an HTTP context test double, you can write unit tests for each route in the routing table. These unit tests adopt the following general phases:

- *Arrange*. In the Arrange phase of the test, you can create a new HTTP context from your test double class. You can set the request URL for this object to be the URL you want to test. You can then create a new route collection and call the **RouteConfig.RegisterRoutes()** method in your web application.
- *Act*. In the Act phase, you can test the routes by calling the **GetRouteData()** method of the route collection. You can then pass the fake HTTP context to this method.
- *Assert*. In the Assert phase, you can use the **RouteData.Values** collection to check that the controller, action and other values are assigned correctly.

In the following example unit test, the default route is checked. The request URL has only a single segment and the test asserts that this segment is passed as the controller name.

A Routing Unit Test

```
        [TestMethod]
        public void Test_Default_Route_ControllerOnly()
        {
            //This test checks the default route when only the controller is specified
            //Arrange
            var context = new FakeHttpContextForRouting(requestUrl: "~/ControllerName");
            var routes = new RouteCollection();
            MyMVCApplication.RouteConfig.RegisterRoutes(routes);
            // Act
            RouteData routeData = routes.GetRouteData(context);
            // Assert
            Assert.AreEqual("ControllerName", routeData.Values["controller"]);
            Assert.AreEqual("Index", routeData.Values["action"]);
            Assert.AreEqual(UrlParameter.Optional, routeData.Values["id"]);
        }
```

Question: You are writing a unit test to check your web application's routing table. You want to ensure that the request is passed to an action method called, "Edit". What line of code would you use in the Assert phase of the test to check this fact?

Lesson 3
Creating a Navigation Structure

A navigation structure is a logical hierarchy of webpages, which enables users to browse to the information that interests them. In simple web applications, the design of the navigation structure is also usually simple. As your web application grows, it becomes difficult to present menus and navigation controls that show the users clearly what their next click should be, at every step. You need to understand how to build menus, which help users understand where they are in your site and where to go next. You can do this by building menus and breadcrumb controls.

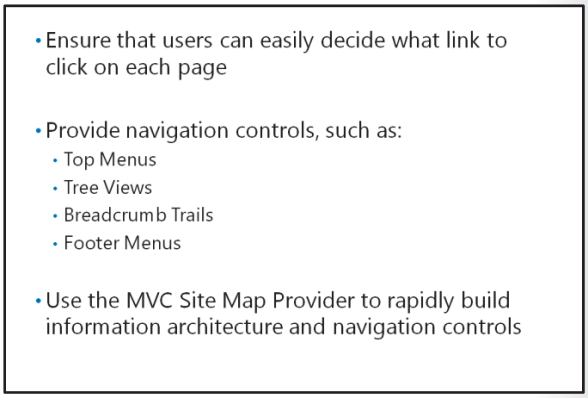
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the importance of well-designed navigation.
- Build clear menus in a web application.
- Add breadcrumb controls to a web application.
- Build site navigation in a web application.

The Importance of Well-Designed Navigation

The information architecture you design for your web application is a logical hierarchy of objects. This hierarchy should be clear and understandable to the audience of your web application. If your web application is designed for people with specialist knowledge, you can implement information architecture that requires specialist knowledge. For example, in an engineering web application, you could base your information architecture on specialist technologies and solutions that engineers understand. However, such information architecture would not be helpful to the general public.



After designing the information architecture, you must present it in your web application. You can add routes to ensure that URLs reflect your information architecture. Similarly, you must add navigation controls to present the information architecture in the body of each webpage. The navigation controls you build are the primary means by which users browse your web application and locate content. You should ensure two things with these navigation controls:

- *An obvious next step*. You should ensure that users can easily decide what link to click, on each page. To achieve this aim, you can consider use cases. For each use case, you can list what the user knows when they arrive at your web application. The user should be able to choose the right link without any specialist or technical knowledge. For example, avoid basing your links on any kind of technical jargon, unless you know that your audience understands it.
- *A clear context*. In a simple web application with only one or two levels in the information architecture, users can easily keep track of where they are. In larger web applications with many levels, you must provide navigation controls, which show users where they are and how to navigate back up the hierarchy. Tree views and breadcrumb controls perform this function well.

Best Practice: Base your navigation controls on the information hierarchy, rather than on the model and model classes, database tables, or other technical structures in your web application. Users will probably not understand the model classes, controllers, views, and data storage mechanisms that underlie your web application.

Common Types of Navigation Controls

Remember that most web application visitors have browsed thousands of Internet applications before they visit your home page. Visitors are usually familiar with common types of menus used on many web applications, and you can make use of this knowledge by using the same types of menus in your web application. Common menu types include the following:

- *Top menus*. Often placed at the top of the page or immediately beneath the application branding, top menus link to the main areas of your web application. In some applications, each item in the top menu has a submenu with links to lower level items and pages.
- *Tree views*. A tree view is a hierarchical menu that can display many items in multiple levels in your web application. By default, a tree view often displays only one or two levels of the information architecture. Users can expand lower levels to find the content they need. Tree views enable users to explore multiple levels on a single page. Tree views can also show clearly the position of the current page, in the hierarchy.
- *Breadcrumb trails*. Often presented as a horizontal sequence of links across the top of a webpage, a breadcrumb trail shows the current page and its parents in all the higher levels of the hierarchy. Breadcrumb trails show the current position of the user, in the context of the web application and enable users to navigate to higher, less specific pages.
- *Footer menus*. The page footer is often not visible when a user arrives on a page. The user must scroll to the end of the page to see the footer. The footer is often used as a location for important, but not business-critical, links. For example, links to copyright and other legal information, links to parent companies, and links to site maps are often placed in the footer menu.

The MVC Site Map Provider

In ASP.NET, one way to rapidly build information architecture and navigation controls to present that architecture is to use a site map provider. A site map provider is an ASP.NET component that stores the logical hierarchy of your web application. Navigation controls such as top menus, tree views, and breadcrumb trails can take their links from this single hierarchy.

The MVC Site Map Provider is a provider designed to work with MVC controllers and actions. It presents the information architecture that you can configure in a simple XML file. This hierarchy is completely independent of the model, controllers, and actions in your applications, although you can build links to them. The provider also includes HTML helpers, which you can use in views to render menus, tree views, and breadcrumb trails.

Note: The MVC Site Map Provider is a component by Maarten Balliauw that is not included in MVC web applications, by default. You can find it in the NuGet package manager by searching for "MvcSiteMapProvider".

Question: Analysis of web logs has shown that visitors to your web application can navigate to low-level pages in your information architecture, quickly and easily. However, they subsequently find other pages by returning to the home page and navigating the entire hierarchy again from the top, or else they use the search tool. How can you enable users to navigate to higher levels without starting from the home page again?

Configuring the MVC Site Map Provider

After installing the MVC Site Map Provider from the NuGet package manager, you must configure the provider and set up the logical hierarchy it should present, before you can render any menus.

Configuring the MVC Site Map Provider in Web.Config

You should configure the MVC Site Map Provider, by adding elements to the Web.config file in the root folder of your MVC web application. When you install the site map provider, the NuGet package adds a **<siteMap>** element to the **<system.web>** section of this file.

The following code example shows how to configure the site map provider.

The <siteMap> Element in Web.config

```
<siteMap defaultProvider="MvcSiteMapProvider" enabled="true">
   <providers>
      <clear />
      <add name="MvcSiteMapProvider"
         type="MvcSiteMapProvider.DefaultSiteMapProvider, MvcSiteMapProvider"
         siteMapFile="~/Mvc.Sitemap"
         securityTrimmingEnabled="false"
         cacheDuration="5"
         enableLocalization="false"
         scanAssembliesForSiteMapNodes="false"
         includeAssembliesForScan=""
         excludeAssembliesForScan=""
         attributesToIgnore=""
         nodeKeyGenerator="MvcSiteMapProvider.DefaultNodeKeyGenerator,
MvcSiteMapProvider"
         controllerTypeResolver="MvcSiteMapProvider.DefaultControllerTypeResolver,
MvcSiteMapProvider"

actionMethodParameterResolver="MvcSiteMapProvider.DefaultActionMethodParameterResolver,
            MvcSiteMapProvider"
         aclModule="MvcSiteMapProvider.DefaultAclModule, MvcSiteMapProvider"
         siteMapNodeUrlResolver="MvcSiteMapProvider.DefaultSiteMapNodeUrlResolver,
            MvcSiteMapProvider"

siteMapNodeVisibilityProvider="MvcSiteMapProvider.DefaultSiteMapNodeVisibilityProvider,
            MvcSiteMapProvider"

siteMapProviderEventHandler="MvcSiteMapProvider.DefaultSiteMapProviderEventHandler,
            MvcSiteMapProvider"
      />
   </providers>
</siteMap>
```

Note that you can configure the name of the XML file where the site map is stored, enable security trimming, and configure caching.

Additional Reading: For complete details of the configuration values you can use with the MVC Site Map Provider, together with other documentation, go to http://go.microsoft.com/fwlink/?LinkID=288963&clcid=0x409

Creating the Site Map File

The MVC Site Map Provider is the only site map provider that enables you to specify a controller and action for every item in the site map. You do this by completing a site map file. By default, the site map file is named Mvc.sitemap and is stored in the highest level of your web application. The hierarchy consists of nested **<mvcSiteMapNode>** elements. This is the hierarchy that the helpers will display in navigation controls.

The following example shows simple information architecture as described in an MVC site map provider site map file.

An MVC Site Map File

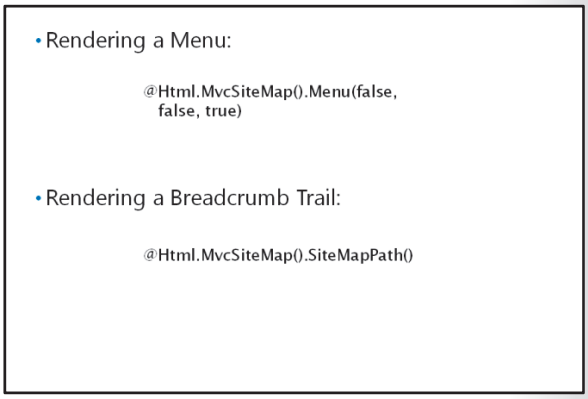
```
<?xml version="1.0" encoding="utf-8" ?>
<mvcSiteMap xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://mvcsitemap.codeplex.com/schemas/MvcSiteMap-File-3.0"
   xsi:schemaLocation=
       "http://mvcsitemap.codeplex.com/schemas/MvcSiteMap-File-3.0 MvcSiteMapSchema.xsd"
   enableLocalization="true">
   <mvcSiteMapNode title="Home" controller="Home" action="Index">
      <mvcSiteMapNode title="Products" controller="Product" action="Index"
key="AllProducts">
         <mvcSiteMapNode title="Bikes" controller="Category" action="Display">
         </mvcSiteMapNode>
      </mvcSiteMapNode>
      <mvcSiteMapNode title="Latest News" controller="Article" action="DisplayLatest" >
      <mvcSiteMapNode title="About Us" controller="Home" action="About" />
    </mvcSiteMapNode>
</mvcSiteMap>
```

Question: You use the MVC Site Map Provider. Your boss notices that an incorrect link appears in the highest menu and the tree view on every page in your web application. Your boss asks you to fix every page in the web application. What steps should you take?

Adding Menu Controls

After configuring the site map provider and creating a site map file, you can add menus, tree views, and other navigation controls to your views. The MVC site map provider includes some HTML helpers to make this easy.

The following example code shows how to render a main menu in an MVC view, by using the **HTML.MvcSiteMap()** helper.



Rendering a Menu Control

```
@Html.MvcSiteMap().Menu(false, false,
true)
```

In this example, the first parameter specifies that the menu does not start from the current node. The menu starts from the top-level node, which is usually the home page of the web application. The second parameter specifies that the starting node does not display as a child node at the same level as second-level nodes. The third parameter specifies that the starting node displays in the menu.

You can also use the **Html.MvcSiteMap()** helper to render a breadcrumb control, as displayed in the following code.

Rendering a Breadcrumb Trail

```
@Html.MvcSiteMap().SiteMapPath()
```

The **Html.MvcSiteMap()** helper also includes functions that you can use to render node titles and complete site maps.

Additional Reading:

For complete reference documentation that details the **Html.MvcSiteMap()** helper and its functions, go to http://go.microsoft.com/fwlink/?LinkID=288964&clcid=0x409

Question: What are some of the benefits of using the **Html.MvcSiteMap()** helper?

Demonstration: How to Build Site Navigation

In this demonstration, you will see how to create menus and breadcrumb trails by using the MVC site map provider and its HTML helpers.

Demonstration Steps

1. On the **PROJECT** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Manage NuGet Packages**.
2. In the **OperasWebSite - Manage NuGet Packages** dialog box, click **Online**.
3. In the **Search Online (Ctrl+E)** box of the **OperasWebSite - Manage NuGet Packages** dialog box, type **mvcsitemapprovider**, and then click the **Search** button.
4. In the **OperasWebSite - Manage NuGet Packages** dialog box, click **Install** corresponding to **MvcSiteMapProvider**.
5. In the **OperasWebSite - Manage NuGet Packages** dialog box, ensure that the **MvcSiteMapProvider** package is installed, and then click **Close**.
6. In the Solution Explorer pane of the **OperasWebSite - Microsoft Visual Studio** window, expand **OperasWebSite**, collapse **App_Start**, collapse **Controllers**, and then collapse **Views**.
7. In the Solution Explorer pane, under Global.asax, click **Mvc.sitemap**.
8. In the Mvc.sitemap code window, locate the following code.

```
<mvcSiteMapNode title="Home" controller="Home" action="Index">
```

9. Place the mouse cursor at the end of the located code, press Enter, and then type the following code.

```
<mvcSiteMapNode title="All Operas" controller="Opera" action="Index" key="AllOperas"
/>
```

10. On the **BUILD** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Build Solution**.
11. In the Solution Explorer pane, expand **Views**, expand **Home**, and then click **Index.cshtml**.
12. In the Index.cshtml code window, place the mouse cursor after the ** ** tag, press Enter, and then type the following code.

```
Menu: @Html.MvcSiteMap().Menu(false, false, true)
```

13. Place the mouse cursor at the end of the site map menu code block, press Enter, and then type the following code.

```
Breadcrumb Trail: @Html.MvcSiteMap().SiteMapPath()
```

14. In the Solution Explorer pane, under Views, expand **Opera**, and then click **Index.cshtml**.
15. In the **Index.cshtml** code window, place the mouse cursor at the end of the **<body>** tag, press Enter, and then type the following code.

```
Menu: @Html.MvcSiteMap().Menu(false, false, true)
```

16. Place the mouse cursor at the end of the site map menu code block, press Enter, and then type the following code.

```
Breadcrumb Trail: @Html.MvcSiteMap().SiteMapPath()
```

17. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.

Note: On the Operas I Have Seen page, ensure that a menu is added.

18. On the Operas I Have Seen page, under Menu, click the **All Operas** link.
19. On the Index page, note that the Main Opera List is displayed.

Note: On the Index page, you can also view the menu.

20. In the Breadcrumb Trail section of the Index page, click the **Home** link.

Note: The Operas I Have Seen page is displayed.

21. On the Operas I Have Seen page, under Menu, click the **About** link.

Note: The About page of the web application is displayed.

22. In the Windows Internet Explorer window, click the **Close** button.
23. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Lab: Structuring ASP.NET MVC 4 Web Applications

Scenario

An important design priority for the Photo Sharing application is that the visitors should be able to easily and logically locate photographs. Additionally, photo galleries and photos need to appear high in search engine results. To implement these priorities, you have been asked to configure routes that enable the entry of user-friendly URLs to access photos.

You have been asked to ensure that the URLs of the following forms work to display a photo:

- ~/photo/display/*PhotoId*. In this form of URL, PhotoId is the database ID of the photo object. This form of URL already works because it matches the default route.
- ~/photo/*PhotoId*. In this form of URL, PhotoId is the database ID of the photo object. This is the logical URL to enter when you know the ID of the photo that you want to access.
- ~/photo/title/*PhotoTitle*. In this form of URL, PhotoTitle is the title of the photo object. This is the logical URL to enter when you know the title of the photo that you want to access.

You have also been asked to implement the following navigation controls in the Photo Sharing application:

- A menu with links to the main site areas
- A breadcrumb control

These navigation controls will be added to the menu after the completion of the main site areas.

Objectives

After completing this lab, you will be able to:

- Add routes to the ASP.NET Routing Engine in an ASP.NET MVC application.
- Build navigation controls within ASP.NET views.

Lab Setup

Estimated Time: 40 minutes

Virtual Machine: **20486B-SEA-DEV11**

Username: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

- On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.
- In the navigation pane of the **Options** dialog box, click **Package Manager**.
- Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Using the Routing Engine

Scenario

In this exercise, you will:

- Create unit tests for the routes you wish to create.
- Add routes to the application that satisfy your tests.
- Try out routes by typing URLs in the Internet Explorer Address bar.

This approach conforms to the principles of Test Driven Development (TDD).

The main tasks for this exercise are as follows:

1. Test the routing configuration.
2. Add and test the Photo ID route.
3. Add and test the Photo Title route.
4. Try out the new routes.

▶ Task 1: Test the routing configuration.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **PhotoSharingApplication** solution from the following location:
   - File location: **Allfiles (D):\Labfiles\Mod07\Starter\PhotoSharingApplication**
3. Add an existing code file to the **Photo Sharing Tests** project, which contains test doubles for HTTP objects, by using the following information:
   - Destination folder: **Doubles**
   - Source folder: **Allfiles (D):\Labfiles\Mod07\Fake Http Classes**
   - Code file: **FakeHttpClasses.cs**
4. Add a reference from the **Photo Sharing Tests** project to the **System.Web** assembly.
5. Add a new **Unit Test** item to the **PhotoSharingTests** project. Name the file, **RoutingTests.cs**.
6. Add **using** statements to the RoutingTests.cs file for the following namespaces:
   - **System.Web.Routing**
   - **System.Web.Mvc**
   - **PhotoSharingTests.Doubles**
   - **PhotoSharingApplication**
7. Rename the **TestMethod1** test to **Test_Default_Route_ControllerOnly**.
8. In the **Test_Default_Route_ControllerOnly** test, create a new **var** by using the following information:
   - Name: **context**
   - Type: **FakeHttpContextForRouting**
   - Request URL: **~/ControllerName**
9. Create a new **RouteCollection** object named **routes** and pass it to the **RouteConfig.RegisterRoutes()** method.
10. Call the **routes.GetRouteData()** method to run the test by using the following information:

- Return type: **RouteData**
- Return object name: **routeData**
- Method: **routes.GetRouteData**
- HTTP context object: **context**

11. Assert the following facts:
    - That **routeData** is not null
    - That the **controller** value in **routeData** is "ControllerName"
    - That the **action** value in **routeData** is "Index"
12. Add a new test to the **RoutingTests** class named, **Test_Photo_Route_With_PhotoID**.
13. In the **Test_Photo_Route_With_PhotoID()** test method, create a new **var** by using the following information:
    - Name: **context**
    - Type: **FakeHttpContextForRouting**
    - Request URL: **~/photo/2**
14. Create a new **RouteCollection** object named **routes** and pass it to the **RouteConfig.RegisterRoutes()** method.
15. Call the **routes.GetRouteData()** method to run the test by using the following information:
    - Return type: **RouteData**
    - Return object name: **routeData**
    - Method: **routes.GetRouteData**
    - Http context object: **context**
16. Assert the following facts:
    - That **routeData** is not null
    - That the **controller** value in **routeData** is "Photo"
    - That the **action** value in **routeData** is "Display"
    - That the **id** value in **routeData** is "2"
17. Add a new test to the **RoutingTests** class named **Test_Photo_Title_Route**
18. In the **Test_Photo_Title_Route** test method, create a new **var** by using the following information:
    - Name: **context**
    - Type: **FakeHttpContextForRouting**
    - Request URL: **~/photo/title/my%20title**
19. Create a new **RouteCollection** object named **routes** and pass it to the **RouteConfig.RegisterRoutes()** method.
20. Call the **routes.GetRouteData()** method to run the test by using the following information:
    - Return type: **RouteData**
    - Return object name: **routeData**
    - Method: **routes.GetRouteData**

- HTTP context object: **context**

21. Assert the following facts:
    - That **routeData** is not null
    - That the **controller** value in **routeData** is "Photo"
    - That the **action** value in **routeData** is "DisplayByTitle"
    - That the **title** value in **routeData** is "my%20title"
22. Run all the tests in the **Photo Sharing Tests** project to verify the test results.

Note: Two of the tests should fail because the routes that they test do not yet exist.

▶ Task 2: Add and test the Photo ID route.

1. Open the **RouteConfig.cs** file in the **PhotoSharingApplication** project.
2. Add a new route to the Photo Sharing application by using the following information. Add the new route before the default route:
    - Name: **PhotoRoute**
    - URL: **photo/{id}**
    - Default controller: **Photo**
    - Default action: **Display**
    - Constraints: **id = "[0-9]+"**
3. Run all the tests in the **Photo Sharing Tests** project to verify the test results.

▶ Task 3: Add and test the Photo Title route.

1. Add a new route to the Photo Sharing application by using the following information. Add the new route after the **PhotoRoute** route but before the default route:
    - Name: **PhotoTitleRoute**
    - URL: **photo/title/{title}**
    - Default controller: **Photo**
    - Default action: **DisplayByTitle**
2. Add a new action method to **PhotoController.cs** by using the following information:
    - Scope: **public**
    - Return type: **ActionResult**
    - Name: **DisplayByTitle**
    - Parameter: a **string** named **title**
3. In the **DisplayByTitle** action method, use the **context.FindPhotoByTitle()** method to locate a photo. If the **context.FindPhotoByTitle()** method returns **null**, return **HttpNotFound()**. Otherwise, pass the photo to the **Display** view.
4. Run all the tests in the **Photo Sharing Tests** project to verify the test results.

▶ Task 4: Try out the new routes.

1. Start the **PhotoSharingApplication** project with debugging.

2. View properties of the **Display** link of any image on the home page, and note the route that has been used to formulate the link.
3. Display any image to verify the URL.
4. Access the following relative URL:
   - **/photo/title/sample photo 3**
5. Stop debugging.

Results: After completing this exercise, you will be able to create a Photo Sharing application with three configured routes that enable visitors to easily locate photos by using logical URLs.

Exercise 2: Optional—Building Navigation Controls

Scenario

In this exercise, you will:

- Add the MVC site map provider to your Photo Sharing application.
- Use the MVC site map provider to create a menu and a breadcrumb control.

At this stage of development, most of the main areas in the Photo Sharing Application are not yet built; therefore, the menu will show only the home page and the All Photos gallery. Your team will add new nodes to the site map as areas of the site are completed.

Complete this exercise if time permits

The main tasks for this exercise are as follows:

1. Install the MVC site map provider.
2. Configure the MVC site map provider.
3. Render menus and breadcrumb trails.
4. Try out the menus.

▶ Task 1: Install the MVC site map provider.

1. Start the **NuGet Packages** manager and locate the **MvcSiteMapProvider** package.
2. Install the MvcSiteMapProvider package.

▶ Task 2: Configure the MVC site map provider.

1. Open the Web.config file in the **PhotoSharingApplication** project.
2. Configure the **MvcSiteMapProvider** to disable localization.
3. Save the changes made to the Web.config file.
4. Open the Mvc.sitemap file and remove the **<mvcSiteMapNode>** element with the title, **About**.
5. Add an **<mvcSiteMapNode>** element within the Home node by using the following information:
   - Title: **All Photos**
   - Controller: **Photo**
   - Action: **Index**
   - Key: **AllPhotos**

6. Save the changes made to the Mvc.sitemap file.
7. Build the solution.

▶ Task 3: Render menus and breadcrumb trails.

1. Render a site menu on the **Home Index** view by using the following information:
   - Helper: **Html.MvcSiteMap()**
   - Method: **Menu**
   - Start From Current Note: **False**
   - Starting Node in Child Level: **False**
   - Show Starting Node: **True**
2. Render a breadcrumb trail on the **Home** view by using the following information:
   - Helper: **Html.MvcSiteMap()**
   - Method: **SiteMapPath**
3. Render a site menu on the **Photo Index** view by using the following information:
   - Helper: **Html.MvcSiteMap()**
   - Method: **Menu**
   - Start From Current Note: **False**
   - Starting Node in Child Level: **False**
   - Show Starting Node: **True**
4. Render a breadcrumb trail on the **Photo Index** view by using the following information:
   - Helper: **Html.MvcSiteMap()**
   - Method: **SiteMapPath**

▶ Task 4: Try out the menus.

1. Start debugging the **PhotoSharingApplication** project.
2. Use the menu option to browse to All Photos.
3. Use the breadcrumb trail to browse to the Home page.
4. Stop debugging and close the Visual Studio application.

Results: After completing this exercise, you will be able to create a Photo Sharing application with a simple site map, menu, and breadcrumb control.

Question: In Exercise 1, when you ran the tests for the first time, why did Test_Default_Route_Controller_Only pass when Test_Photo_Route_With_PhotoID and Test_Photo_Title_Route fail?

Question: Why is the constraint necessary in the **PhotoRoute** route?

Module Review and Takeaways

To create a compelling web application that is easy to navigate, you must consider carefully the architecture of the information you present. You should try to ensure that the hierarchy of objects you present in URLs and navigation controls matches user expectations. You should also ensure that users can understand this hierarchy of objects without specialist technical knowledge. By using routes and site map providers, you can present logical information architecture to application visitors.

Best Practice: The default route is logical. However, it requires knowledge of controllers and actions for users to understand URLs. You can consider creating custom routes that can be understood with information that users already have.

Best Practice: You can use breadcrumb trails and tree view navigation controls to present the current location of a user, in the context of the logical hierarchy of the web application.

Best Practice: You can use unit tests to check routes in the routing table, similar to the manner with which you use tests to check controllers, actions, and model classes. It is easy for a small mistake to completely change the way URLs are handled in your web application. This is because new routes, if poorly coded, can override other routes. Unit tests highlight such bugs as soon as they arise.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
A route never takes effect.	

Review Question(s)

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
You have implemented the MVC Site Map Provider in your web application and used it to build menus and breadcrumb trails with which users can navigate the logical hierarchy. MVC automatically takes routes from the MVC Site Map Provider so the same hierarchy is used in URLs.	

Question: You want to ensure that when the user specifies a relative URL in the form, "customer/3546", the request is forwarded to the **DisplayByID()** action in the **CustomerController**. You also want to ensure that when the user specifies a relative URL in the form, "customer/*fullname*", the request is forwarded to the **DisplayByName()** action in the **CustomerController**. What routes should you add?

Module 08

Applying Styles to ASP.NET MVC 4 Web Applications

Contents:

Module Overview

While building web applications, you should apply a consistent look and feel to the application. You should include consistent header and footer sections in all the views. ASP.NET MVC 4 includes features such as cascading style sheets (CSS) styles and layouts that enhance the appearance and usability of your web application. ASP.NET MVC 4 also includes features such as mobile-specific views that allow you to build applications for different browsers or mobile devices.

Objectives

After completing this module, you will be able to:

- Apply a consistent layout to ASP.NET MVC 4 applications by using CSS and layout.
- Develop device-specific views to support various browsers.
- Build adaptive user interface for mobile devices.

Lesson 1
Using Layouts

You need to build multiple views to support the operations of the application, such as creating an order and querying order history. However, several maintenance issues arise while changing the common part of the application layout, because of which you need to update each view. To resolve these maintenance issues, you can build a common module or a shared view. A shared view that helps to store the application logic is called a layout. ASP.NET MVC 4 includes features that help simplify the process of creating and using layouts. You can further simplify the application management process by using the _ViewStart file, to apply the layout to each view, instead of individually editing each view.

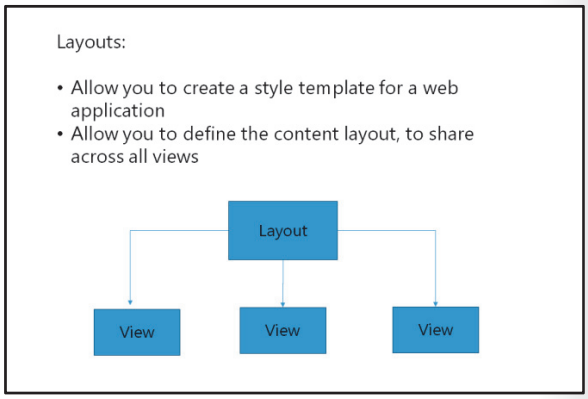
Lesson Objectives

After completing this lesson, you will be able to:

- Describe layouts.
- Describe how to use layouts.
- Describe the _ViewStart file.

What Are Layouts?

The ASP.NET MVC 4 Razor engine includes a feature called layouts. Layouts are also called template views. Layouts enable you to define a common style template, and then apply it to all the views in a web application. The functionality of layouts is similar to that of the master page in a traditional ASP.NET web application. You can use layouts to define the content layout or logic that is shared across views.



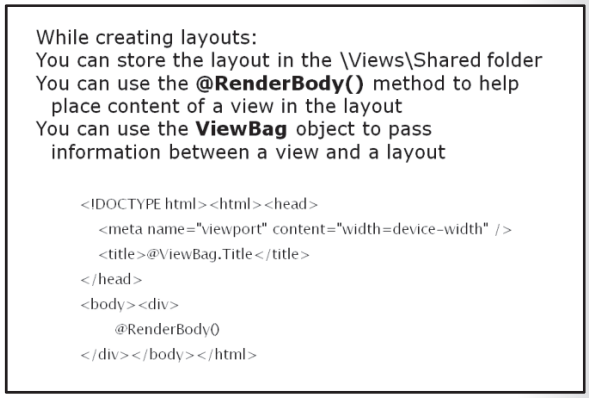
You can define multiple layouts in an ASP.NET MVC 4 application, and each layout can have multiple sections. You can define these sections anywhere in the layout file, even in the **<head>** section of the HTML. Sections enable you to output dynamic content to multiple, non-contiguous, regions of the final response.

Question: What are some common scenarios when you would use layouts?

Creating a Layout

While creating layouts, you need to store the layout files in the \Views\Shared folder of the project. The \Views\Shared folder is the default location, where you can store common view files or templates.

The following example illustrates a layout.



A Layout View

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport"
content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html
```

In the preceding example, the **@RenderBody()** method indicates to the rendering engine where the content of the view goes.

ViewBag is an object that the layout and view shares. You can use the **ViewBag** object to pass information between a view and a layout. To pass information, you need to add a property to the **ViewBag** object, in the ViewPage of the ViewController or View file, and use the same property in the layout file. Properties help you control the content in the layout, to dynamically render webpages from the code in the view file. For example, consider that the template uses the **ViewBag.Title** property to render the **<title>** content in the view. This property helps define the **Title** property of the **ViewBag** object in the view and retrieve the property in the layout. This retrieval is possible because the code in the view file runs before the layout runs.

The following example illustrates a layout that contains a section.

Using Sections in Layouts

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div id="menu">
        @RenderSection("MenuBar",required:false)
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

The **MenuBar** parameter in the **RenderSection()** helper method specifies the name of the section that you want to render in the layout. The **required** parameter is optional; it allows you to determine if the section you render is required. Consider that a section is required, and you do not implement the section in the layout file. In this case, ASP.NET MVC 4.0 displays the **Section not defined** exception at runtime.

Implementing sections in a layout file makes it easier to track content errors. If a section is not required, you can choose to not include it in the layout.

Additional Reading: For more information about layouts and sections, you can see: http://go.microsoft.com/fwlink/?LinkID=288965&clcid=0x409

Question: Why do you have multiple sections in a layout?

Linking Views and Layouts

After defining the layout, you should link the layout to the view files. You should first remove the content that is not required anymore in the view. Then, you need to create the link between the view and the layout, so that the content removed from the view is not reflected in the layout.

To link a layout to a view, you need to add the **Layout** directive at the top of the view file. The following code shows how to add a layout in a view file.

To link views and layouts:
- You can add the **Layout** directive at the top of the view file
- You can use the **ViewBag** object:
  - Use properties to pass information between views and templates
  - Use the **@section** directive to define sections in the layout
- You can use the **_ViewStart** file to define the layout
  - Add the _ViewStart.cshtml file in the \Views folder of your project

Linking to a View

```
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/SiteLayout.cshtml";
}
<h2>Details</h2>
```

You can use the **ViewBag.Title** property to pass page title information from the view to the layout. You can define other properties along with the **ViewBag** object, such as **<meta>** elements in the **<head>** section, and enable them to pass information to the layout.

If you have multiple sections in the layout file, you define the content of the sections by using the **@section** directive. The following code illustrates how to use the **@section** directive.

Using the @section Directive

```
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/SiteLayout.cshtml";
}
<h2>Details</h2>
@section MenuBar {
    <p> this is menu</p>
}
```

In the preceding example, you set the layout file to display sections at the top of each view file. Usually, you have the same layout across the entire web application or section. You can define the layout for an application or section, by using the **_ViewStart** file. During runtime, the code in the **_ViewStart** file runs before all the other views in the web application. Therefore, you can place all common application logics in the **_ViewStart** file.

To use the **_ViewStart** file, you need to add the _ViewStart.cshtml file in the \Views folder of your project. The following code illustrates the contents of the _ViewStart file.

The _ViewStart File

```
@{
    Layout = "~/Views/Shared/SiteLayout.cshtml";
}
```

Question: When should you use the **_Viewstart** file?

Lesson 2
Applying CSS Styles to an MVC Application

Cascading Style Sheets (CSS) is an industry standard for applying styles to HTML pages. Different methods of applying CSS to a webpage are available. These methods include external CSS file, inline CSS, and CSS code block in HTML. Developers usually use an external CSS file, because this file is shared across multiple pages and it helps apply a consistent style across the application. You need to know how to import styles into a web application, to ensure consistency in the appearance of the application. You also need to know how to use Expression Blend to create rich and interactive user interfaces.

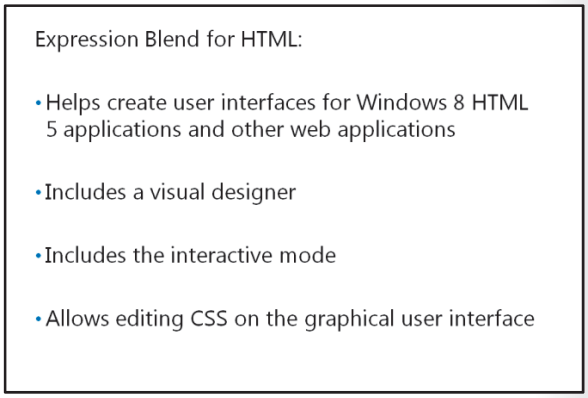
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the user experience that can be accomplished by using Expression Blend.
- Import styles into an ASP.NET MVC 4 web application.
- Apply a consistent look and feel to an MVC 4 web application.

Overview of User Interface Design with Expression Blend

Expression Blend is a visual authoring tool that helps developers and designers to create user interfaces for Windows 8 HTML5 applications and other web applications. Expression Blend for HTML includes a visual designer, which serves as a good complement to Visual Studio. Visual Studio is developer-centric, whereas Expression Blend for HTML is designer-centric. The functions in Expression Blend facilitate developing HTML styles and CSS.

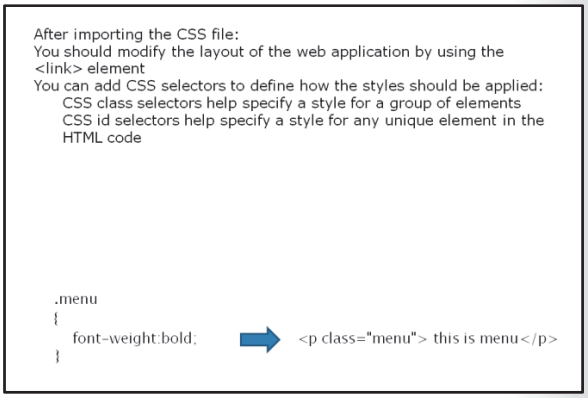


Expression Blend for HTML includes the interactive mode that enables developers to visualize the result of the design, HTML, CSS, and JavaScript, by displaying results on the screen while they edit the code. Expression Blend for HTML also includes CSS tools that enable you to generate and edit CSS styles on the graphical user interface.

Question: What are the key benefits of using Expression Blend for HTML to edit CSS?

Importing Styles into an MVC Web Application

After creating CSS styles, you should import these styles into the web application. After importing the CSS file into the web application, you need to modify the layout of the web application, so that you can use the CSS styles that you imported. You can modify the layout of a web application, by using the **<link>** element.



The following example shows how to use the **<link>** element.

Linking to a Style Sheet

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Views/Shared/StyleSheet1.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="menu">
        @RenderSection("MenuBar",required:false)
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

CSS selectors help browsers to determine how the CSS styles should be applied. You can use various selectors, such as class and id selectors, to apply styles to HTML elements.

CSS class Selector

You can define a CSS class selector to specify a style for a group of elements. To apply the class selector to an HTML element, you need to add the **class** attribute to the HTML element. You can use the **.<class>** syntax to add the style in the CSS file.

The following example shows how to add the class selector in a view.

Using a Class

```
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/SiteLayout.cshtml";
}
<h2>Details</h2>
@section MenuBar {
    <p class=”menu”> this is menu</p>
}
```

The following CSS snippet shows how to create the class selector.

Applying a Style to a Class

```
.menu
{
    font-weight:bold;
}
```

CSS id Selector

You can use the CSS id selector to specify a style for any unique element in your HTML code. To apply the id selector to an HTML element, you need to add the **id** attribute and a unique name to the HTML element. You can use the **#<id>** syntax to add the style in the CSS file.

The following example shows how to use the id attribute in a view.

Using The id Selector

```
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/SiteLayout.cshtml";
}
<h2>Details</h2>
@section MenuBar {
    <p id="leftmenu"> this is menu</p>
}
```

The following CSS snippet shows how to create the CSS id selector.

Creating an ID Style

```
#leftmenu
{
    font-size:16px;
}
```

Question: What are some common scenarios when you would use the class selector? What are some common scenarios when you would use the id selector?

Demonstration: How to Apply a Consistent Look and Feel

In this demonstration, you will see how to:

- Create a layout.
- Apply the layout to an MVC view.

Demonstration Steps

1. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.

Note: On the Operas I Have Seen page, note that the main heading, the menu list, and the breadcrumb control are displayed.

2. On the Operas I Have Seen page, click the **All Operas** link.

Note: On the localhost page, the main heading, the menu list, and the breadcrumb controls are not displayed.

3. On the localhost page, click the **Details** link corresponding to any image.

Note: On the localhost page, the details of the opera are displayed. The main heading, the menu list, and the breadcrumb controls are not displayed.

4. In the Windows Internet Explorer window, click the **Close** button.
5. In the Solution Explorer pane, expand **OperasWebSite**, and then expand **Views**.
6. In the Solution Explorer pane, under Views, right-click **Shared**, point to **Add**, and then click **View**.
7. In the **View name** box of the **Add View** dialog box, type **SiteTemplate**.
8. In the **View engine** box, ensure that the value is **Razor (CSHTML)**, and then ensure that the **Create a strongly-typed view** check box is cleared.
9. In the **Add View** dialog box, clear the **Use a layout or master page** check box, and then click **Add**.
10. In the **_SiteTemplate.cshtml** code window, locate the following code, select the code, and then press Delete.

```
@{
   Layout = null;
}
```

11. In the **_SiteTemplate.cshtml** code window, locate the following code.

```
<title>_SiteTemplate</title>
```

12. Replace the **TITLE** element with the following code.

```
<title>@ViewBag.Title</title>
```

13. In the Solution Explorer pane, under Views, expand **Home**, and then click **Index.cshtml**.
14. In the Index.cshtml code window, locate the following code, and then select the code.

```
<h1>Operas I Have Seen</h1>
<div class="topmenu">
   @Html.MvcSiteMap().Menu(false, true, true)
</div>
<div class="clear-floats" />
<div class="breadcrumb">
  Breadcrumb Trail: @Html.MvcSiteMap().SiteMapPath()
```

15. On the **EDIT** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Cut**.
16. In the Solution Explorer pane, under Shared, click **_SiteTemplate.cshtml**.
17. In the _SiteTemplate.cshtml code window, place the mouse cursor in the **DIV** element.
18. On the **EDIT** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Paste**
19. In the _SiteTemplate.cshtml code window, place the mouse cursor at the end of the code you just pasted, press Enter, and then type the following code.

```
<div>
   @RenderBody()
</div>
```

20. Place the mouse cursor after the **</title>** tag, press Enter, and then type the following code.

```
<link type="text/css" rel="stylesheet" href="~/content/OperasStyles.css" />
```

21. In the Solution Explorer pane, under Home, click **Index.cshtml**.
22. In the Razor code block of the Index.cshtml code window, locate the following code, select the code, and then press Delete.

```
Layout = null;
```

23. In the Razor code block, type the following code.

```
ViewBag.Title = "Operas I Have Seen";
```

24. In the Index.cshtml code window, locate the following code, select the code, and then press Delete.

```
<!DOCTYPE html>
<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <title>Operas I Have Seen</title>
</head>
<body>
   <div>
   </div>
```

25. In the Index.cshtml code window, locate the following code, select the code, and then press Delete.

```
   </div>
</body>
</html>
```

26. In the Solution Explorer pane, right-click **Views**, point to **Add**, and then click **View**.
27. In the **View name** box of the **Add View** dialog box, type **_ViewStart**, and then, in the View engine box, ensure that the value is **Razor (CSHTML)**.
28. In the **Add View** dialog box, ensure that the **Create a strongly-typed view** and the **Use a layout or master page** check boxes are cleared, and then click **Add**.
29. In the _ViewStart.cshtml code window, locate the following code.

```
Layout = null;
```

30. Replace the code with the following code.

```
Layout = "~/Views/Shared/_SiteTemplate.cshtml";
```

31. In the _ViewStart.cshtml code window, locate the following code.

```
<!DOCTYPE html>
```

32. In the _ViewStart.cshtml code window, select from the code located to the end tag of the HTML element, and then press Delete
33. On the **DEBUG** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Start Debugging**.
34. On the Operas I Have Seen page, note the main heading, the menu list, and the breadcrumb control.
35. On the Operas I Have Seen page, click the **All Operas** link, and then, on the Index of Operas page, note that the main heading, the menu list, and the breadcrumb controls are displayed.
36. On the Index of Operas page, click the **Details** link corresponding to any image, and then note that the main heading, the menu list, and the breadcrumb controls are displayed along with the opera details.
37. In the Windows Internet Explorer window, click the **Close** button.
38. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Lesson 3
Creating an Adaptive User Interface

ASP.NET MVC 4 applications facilitate adaptive user interface to render content on different devices. Adaptive user interface is a type of user interface that renders content based on the capability of the target web browser or device. You need to ensure that your application supports mobile devices, so that it reaches all types of audience. You also need to know how to use media queries, mobile-specific views, and jQuery Mobile to ensure that your application is effective on mobile devices.

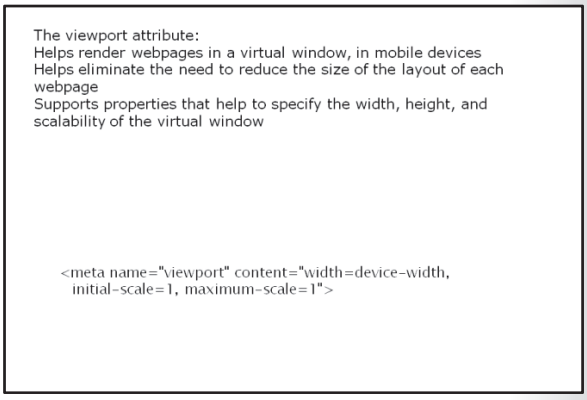
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the HTML5 **viewport** attribute.
- Explain how CSS media queries apply specific CSS based on the capabilities of the browser.
- Explain how you can use MVC 4 templates to render views based on mobile device screen size.
- Describe the use of jQuery Mobile for building user interfaces that work on a wide variety of mobile devices.
- Use NuGet to add jQuery Mobile to your projects.

The HTML5 Viewport Attribute

Adaptive rendering allows you to customize your web application to display differently, based on the capabilities of the web browser or device.



Additional Reading: For more information about adaptive rendering techniques, go to: http://go.microsoft.com/fwlink/?LinkID=288966&clcid=0x409

Mobile browsers such as Internet Explorer use the **viewport** attribute to render webpages in a virtual window. This virtual window is usually wider than the application screen. The **viewport** attribute helps eliminate the need to reduce the size of layout of each page. Reducing the size of the layout can break or distort the display of non-mobile-optimized web applications. Creating the application interface by using the **viewport** attribute enables users to zoom into the different areas of a webpage.

Additional Reading: For more information about the **viewport** attribute, visit: http://go.microsoft.com/fwlink/?LinkID=288967&clcid=0x409

The **viewport** tag is a meta tag that helps to control the width and height of webpages, while it renders to web browsers.

The following example illustrates how to use the **viewport** tag.

Using the Viewport Tag

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
```

The **width** and **height** properties help to specify the width and height of the virtual viewport window. You can specify the width in pixels. You can use the keyword **device-width** to enable the content to fit the native screen size of the browser.

The **initial-scale** property controls the initial scale or zoom level of the webpage. The **maximum-scale**, **minimum-scale**, and **user-scalable** properties control the other scalability features of the webpage.

Question: How can you control the size of the virtual viewport window?

CSS Media Queries

You may sometimes need to apply different CSS styles in your application, to support different browsers. HTML5 includes CSS media queries, which are special selectors that begin with **@media**. Media queries allow conditional application of CSS styles, based on the device conditions or browser capabilities. You can apply media queries in CSS and HTML.

The following example illustrates a media query in CSS.

Characteristics of media queries:
Media queries are special selectors that begin with @media
You can also apply media queries in <link> elements
Media queries support properties that allow you to specify the size details of the targeted display area

```
@media only screen and (max-width: 500px) {
    header{
      float: none;
    }
}
```

Using a Media Query

```
@media only screen and (max-width: 500px) {
    header{
      float: none;
    }
  }
```

You can also apply a media query in the **<link>** element. The following example illustrates how to include a media query in a **<link>** element.

Using a Media Query in the Link Element

```
<link rel="stylesheet" type="text/css" href="smallscreen.css" media="only screen and
(max-width: 500px)" />
```

You can use CSS media queries to apply CSS styles when the screen size is less than 500 pixels. However, you can use CSS media queries only for the screen layout, but not the print layout.

Additional Reading: For more information about media queries, visit http://go.microsoft.com/fwlink/?LinkID=288968&clcid=0x409

The following table describes all properties that you can include in a media query.

Property	Description
width	The width of the targeted display area, which includes the browser window in desktop and mobile devices. In desktop computers, when you resize the browser window, the width of the browser changes. However, on most mobile browsers, you cannot resize the browser window. This implies that the width of the browser remains constant.

Property	Description
height	The height of the targeted display area, which includes the browser window in desktop and mobile devices.
device-width	The width of the entire screen of a device. For a desktop with a screen resolution of 1,024x768, the device-width is usually 1,024 pixels.
device-height	The height of the entire screen of a device. For a desktop with a screen resolution of 1,024x768, the device-height is usually 768 pixels.
orientation	The orientation of the device. If the device-width is larger than **the device-height**, the orientation value is set to **landscape**; otherwise, it is set to **portrait**.
aspect-ratio	The ratio of the **width** and **height** properties.

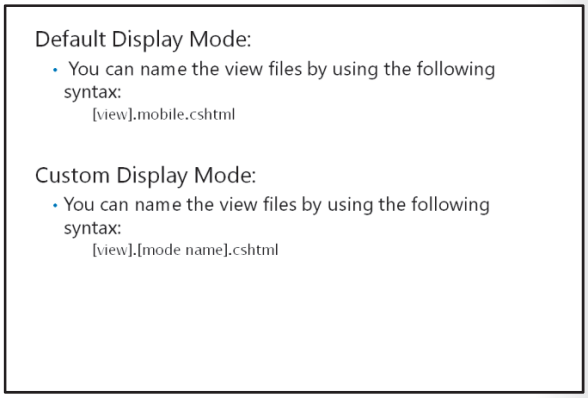
device-aspect-ratio	The ratio of the **device-width** and **device-height** properties. The following example illustrates the device-aspect-ratio for a device with a screen resolution of 1,280x720.
	`@media screen and (device-aspect-ratio: 16/9) { }`
	`@media screen and (device-aspect-ratio: 1280/720) { }`
	`@media screen and (device-aspect-ratio: 2560/1440) { }`

color	The number of bits per color component of the device. If the device is not a color device, the value is zero.
color-index	The number of entries in the color lookup table, of the output device.
monochrome	The number of bits per pixel in a monochrome frame buffer. For non-monochrome devices, this value is zero.
resolution	The resolution of the output device, or the density of the pixels. The common units for this property include dpi and dpcm.
scan	The scanning process of TV output devices.
grid	The property that detects whether the output is in the grid or bitmap format. Grid-based devices return a value of one; all other devices return a value of zero.

Question: Why would you choose to use CSS media queries, instead of using C# code, to define styles for specific browsers?

MVC 4 Templates and Mobile-Specific Views

ASP.NET MVC 4 includes two new features, mobile display mode and custom display mode, which help you to create webpages for mobile devices and different browsers.



Additional Reading: For more information about creating webpages for mobile device browsers, go to http://go.microsoft.com/fwlink/?LinkID=288966&clcid=0x409

ASP.NET MVC 4 enables you to override views for mobile devices by using a different set of view files, rather than using a configuration. When ASP.NET MVC 4 receives a request from a mobile browser, it analyses the request for views with the naming convention **[view].mobile.cshtml**. If ASP.NET MVC 4 detects a view with the mentioned naming convention, ASP.NET MVC 4 will serve the request by using the mobile version of the view; otherwise, it returns the request to the standard view.

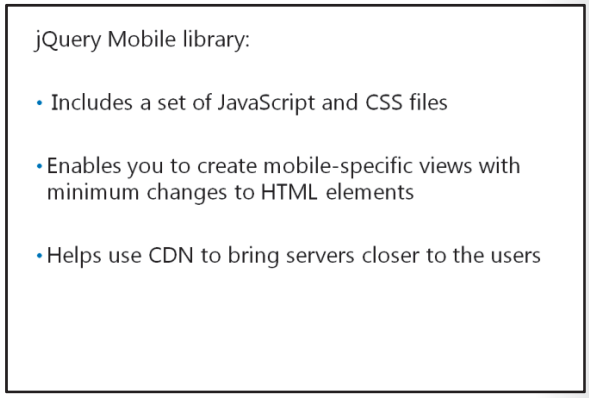
Consider that your web application includes a layout that is specific to a browser. In this case, you can create browser-specific views for that browser by checking the **UserAgent** string of that browser. The **UserAgent** string helps identify a browser.

Additional Reading: For more information about creating browser-specific view, visit http://go.microsoft.com/fwlink/?LinkID=288969&clcid=0x409

Question: Why would you choose device-specific display modes over CSS media queries?

jQuery Mobile

The jQuery Mobile library is a mobile version of the jQuery library, which helps to develop applications that run on mobile devices. jQuery helps to add responsive UI elements to a web application. jQuery also enables developers to add interactive content to the mobile version of a web application. The library also helps to ensure that older mobile devices see usable controls. Similar to jQuery, jQuery Mobile library includes a set of JavaScript and CSS files that enable you to build mobile-specific applications, without adjusting the HTML elements.



jQuery mobile includes the following JavaScript and CSS files:

- jquery.mobile.structure-<version>.css
- jquery.mobile.structure-<version>.min.css
- jquery.mobile.theme-<version>.css

- jquery.mobile.theme-<version>.min.css
- jquery.mobile-<version>.css
- jquery.mobile-<version>.js
- jquery.mobile-<version>.min.css
- jquery.mobile-<version>.min.js
- images/ajax-loader.gif
- images/ajax-loader.png
- images/icons-18-black.png
- images/icons-18-white.png
- images/icons-36-black.png
- images/icons-36-white.png

The jQuery.Mobile.MVC NuGet package simplifies the process of adding the jQuery Mobile library to a web application by eliminating the need to manually install the library files. NuGet is a Visual Studio extension that enables users to download packaged content from the Internet and directly install the content into the project. The NuGet package contains the following items:

- The App_Start\BundleMobileConfig.cs file. This file helps to reference the jQuery JavaScript and CSS files.
- jQuery Mobile CSS files
- A ViewSwitcher controller widget
- jQuery Mobile JavaScript files
- A jQuery Mobile-styled layout file
- A view-switcher partial view. This provides a link at the upper-end of each page; this link helps switch from desktop view to mobile view, and vice versa.
- .png and .gif image files

Additional Reading: For more information about the NuGet package, visit http://go.microsoft.com/fwlink/?LinkID=288970&clcid=0x409

Note: You can use the Microsoft Ajax Content Delivery Network (CDN) to attend to users, by using the servers that are located geographically closer to them. CDN helps to improve the performance of the application. After adding the NuGet package to your ASP.NET MVC application, you can change the code in HTML or the view, to reference the jQuery Mobile library that is hosted on Microsoft Ajax CDN.

Additional Reading: For more information about jQuery mobile that is hosted on Microsoft CDN, visit http://go.microsoft.com/fwlink/?LinkID=288971&clcid=0x409

Question: What are the benefits of using Microsoft Ajax CDN?

Lab: Applying Styles to MVC 4 Web Applications

Scenario

You have created a good amount of the photo-handling functionality for the Photo Sharing web application. However, stakeholders are concerned about the basic black-and-white appearance of the application. In addition, titles and menus do not appear on every page.

To resolve these issues, your manager has asked you to implement the following user interface features:

- *A layout for all webpages*. The layout should include common elements, such as the main menu and breadcrumb controls, which should appear on every page of the application.
- *A style sheet and images for all webpages*. The web design team has provided an HTML mock-up application to show how the final product should look. This mock-up includes a style sheet and image files. You need to import these files and apply them to every page of the application.
- *A mobile-specific view*. The web application should be accessible from mobile devices such as mobile phones and tablets. In particular, you need to ensure that devices with narrow screens can access photos easily.

Objectives

After completing this lab, you will be able to:

- Apply a consistent look and feel to the web application.
- Use layouts to ensure common interface features, such as the headers, are consistent across the entire web application.
- Ensure that the web application renders smoothly on screens of different sizes and aspect ratios.

Lab Setup

Estimated Time: 40 minutes

Virtual Machine: **20486B-SEA-DEV11**

Username: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the MSL-TMG1 virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

a. On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.

b. In the navigation pane of the **Options** dialog box, click **Package Manager**.

c. Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Creating and Applying Layouts

Scenario

In this exercise, you will:

- Browse through the Photo Sharing web application without a layout applied.
- Create a new layout and link the application to the view by using a _ViewStart.cshtml file.
- Modify the home index and photo display views to use the new layout.

- Browse through the resulting web application.

The main tasks for this exercise are as follows:

1. Open and browse through the Photo Sharing application.
2. Create a new layout.
3. Set the default layout for the application.
4. Update the views to use the layout.
5. Browse through the web application.

▶ Task 1: Open and browse through the Photo Sharing application.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **PhotoSharingApplication** solution from the following location:
   - File location: **Allfiles (D):\Labfiles\Mod08\Starter\PhotoSharingApplication**
3. Start the web application in debugging mode and verify that the menu and the breadcrumb trail are available on the home page.
4. Browse to the All Photos webpage and verify that the menu and the breadcrumb trail are not available on this page.
5. Browse to the Sample Photo 1 webpage and verify that the menu and the breadcrumb trail are not available on this page.
6. Stop debugging.

▶ Task 2: Create a new layout.

1. Add a new layout to the **PhotoSharingApplication** project by using the following information:
   - File location: **/Views/Shared**
   - View name: **_MainLayout**
   - View type: **None**
   - Partial view: **None**
   - Layout or master page: **None**
2. Change the content of the **TITLE** element so that the page takes its title from the **ViewBag.Title** property.
3. Add an **H1** heading to the page body by using the following information:
   - Class attribute: **site-page-title**
   - Content: **Adventure Works Photo Sharing**
4. Add a **DIV** element to the page with the class, **clear-floats**.
5. Add a **DIV** element to the page with the id **topmenu**. Within this element, render the main menu for the page by using the following information:
   - Helper: **Html.MvcSiteMap()**

- Method: **Menu()**
- Start from current node: **False**
- Starting node in child level: **True**
- Show starting node: **True**

6. Add a **DIV** element to the page with the id **breadcrumb**. Within this element, render the breadcrumb trail for the page by using the following information:
   - Helper: **Html.MvcSiteMap()**
   - Method: **SiteMapPath()**
7. Add a **DIV** element to the page. Within this element, render the view body by using the following information:
   - Helper: **RenderBody()**
8. Save the layout.

▶ Task 3: Set the default layout for the application.

1. Add a new view to the web application by using the following information:
   - File path: **/Views**
   - View name: **_ViewStart**
   - View type: **None**
   - Partial view: **None**
   - Layout or master page: **None**
2. In the **_ViewStart.cshtml** file, set the **Layout** to **~/Views/Shared/_MainLayout.cshtml**
3. Remove all the HTML code from the **_ViewStart.cshtml** file, except the layout element.
4. Save the file.

▶ Task 4: Update the views to use the layout.

1. Open the Views/Home/Index.cshtml view file.
2. In the first Razor code block, remove the existing line of code, and set the **ViewBag.Title** property to **Welcome to Adventure Works Photo Sharing**.
3. Remove the following:
   - Tags along with the corresponding closing tags:
     - **<!DOCTYPE>**
     - **<html>**
     - **<head>**
     - **<meta>**
     - **<title>**
     - **<body>**
     - ** **
   - Content:
     - **Menu:**

- **Current Location:**

4. Save the changes made to the **Index.cshtml** file.
5. Open the **Views/Photo/Display.cshtml** view file.
6. In the first Razor code block, remove the existing line of code and set the **ViewBag.Title** property to the **Title** property of the **Model** object.
7. Remove the following tags along with the corresponding closing tags:
   - **<!DOCTYPE>**
   - **<html>**
   - **<head>**
   - **<meta>**
   - **<title>**
   - **<body>**
   - ** **
8. Save the changes made to the **Display.cshtml** file.
9. Open the **Views/Shared/Error.cshtml** view file.
10. In the Razor code block, remove the existing line of code and set the **ViewBag.Title** property to **Custom Error**.
11. Remove the following tags along with the corresponding closing tags:
    - **<!DOCTYPE>**
    - **<html>**
    - **<head>**
    - **<meta>**
    - **<title>**
    - **<body>**
    - ** **
12. Save the changes made to the **Error.cshtml** file.

▶ Task 5: Browse through the web application.

1. Start the web application in debugging mode and verify the menu and the breadcrumb trail on the home page.
2. Browse to the **All Photos** webpage and verify that the site title, menu, and breadcrumb trail are available on this page.
3. Browse to **Sample Photo 1** webpage and verify that the site title, menu, and breadcrumb trail are available on this page.
4. Stop debugging.

Results: After completing this exercise, you will be able to create an ASP.NET MVC 4 web application that uses a single layout to display every page of the application.

Exercise 2: Applying Styles to an MVC Web Application

Scenario

In this exercise, you will

- Examine a mockup web application that shows the look-and-feel the web designers have created for the Photo Sharing application.
- Import a style sheet, with the associated graphic files from the mockup application, to your web application, and then update the HTML element classes to apply those styles to the elements in views.

Examine the changes to the user interface after the styles have been applied.

The main tasks for this exercise are as follows:

1. Examine the HTML mockup web application.
2. Import the styles and graphics.
3. Update the element classes to use the styles.
4. Browse the styled web application.

▶ Task 1: Examine the HTML mockup web application.

1. Open the mockup web application and verify the layout of the home page by using the following information:

- File path: **Allfiles (D):\Labfiles\Mod08 \Expression Web Mock Up\default.html**

2. Browse to the **All Photos** webpage and verify the layout of the page.
3. Browse to the details of any photo and verify the layout of the page.
4. Close Internet Explorer.

▶ Task 2: Import the styles and graphics.

1. Add a new top-level folder to the **PhotoSharingApplication** project with the following information:
   - Name of the folder: **Content**
2. Navigate to **Allfiles (D):\Labfiles\Mod08\Expression Web Mock Up\Content**, and add the following existing files to the new folder:
   - PhotoSharingStyles.css
   - BackgroundGradient.jpg
3. Add a **<link>** element to the **_MainLayout.cshtml** file to link the new style sheet by using the following information:
   - Type: **text/css**
   - Relation: **stylesheet**
   - Href: **~/content/PhotoSharingStyles.css**
4. Save the _MainLayout.cshtml file.

▶ Task 3: Update the element classes to use the styles.

1. Open the _PhotoGallery.cshtml file.
2. Locate the first **DIV** element in the file and set the **class** attribute to **photo-index-card**.

3. For the **<img>** tag, remove the **width** attribute and set the **class** attribute to **photo-index-card-img**.
4. For the next **DIV** element, set the class to **photo-metadata**.
5. For the Created By: element, set the class attribute to display-label.
6. For the @Html.DisplayFor(model => item.UserName) element, set the class attribute to display-field.
7. For the Created On: element, set the class attribute to display-label.
8. For the @Html.DisplayFor(model => item.CreatedDate) element, set the class attribute to display-field.

▶ Task 4: Browse the styled web application.

1. Start the web application in debugging mode to examine the home page with the new style applied.
2. Browse to **All Photos** to examine the page with the new style applied.
3. Display a photo of your choice to examine the new style applied.
4. Stop debugging.

Results: After completing this exercise, you will be able to create a Photo Sharing application with a consistent look and feel.

Exercise 3: Optional—Adapting Webpages for Mobile Browsers

Scenario

In this exercise, you will:

- Create a new layout for mobile devices.
- Add a media query to the web application style sheet to ensure that the photo index is displayed on small screens.
- Test the settings applied to the application by using a small browser and changing the user agent string.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Test the application as a mobile device.
2. Add a new mobile layout.
3. Add a media query to the style sheet.
4. Retest the application as a mobile device.

▶ Task 1: Test the application as a mobile device.

5. Start the web application in debugging mode.
1. Resize the browser window to the following dimensions:
   - Width: **480 pixels**
   - Height: **700 pixels**
2. Set the **user agent string** to **IE9 for Windows Phone 7**.

3. Refresh the home page and examine the mobile view of the application.
4. Stop debugging.

▶ Task 2: Add a new mobile layout.

1. Create a copy of the **_MainLayout.cshtml** file in the **Views/Shared** folder and rename the file as **_MainLayout.Mobile.cshtml**.
2. In the **_MainLayout.Mobile.cshtml** file, in the main page heading, place a **
** tag after the words **Adventure Works**.
3. After the **H1** element, add an **H2** element.
   - Content: **Mobile Site**
4. Save the _MainLayout.Mobile.cshtml mobile view.

▶ Task 3: Add a media query to the style sheet.

1. Open the **PhotoSharingStyles.css** style sheet.
2. Add a media query to the style sheet that applies only to screen size and only when the maximum screen width is 500 pixels or less.
3. Examine the existing style of the **topmenulink** class.
4. Add the same style to the media query.
5. In the media query, set the **width** attribute for the **topmenulink** style to 100 pixels.

▶ Task 4: Retest the application as a mobile device.

1. Start the web application in a debugging mode.
2. Clear the browser cache to ensure that the style sheet is reloaded.
3. Set the user agent string to IE9 for Windows Phone 7.
4. Close the developer window and refresh the web application to examine whether the problem persists in the mobile view of the application.
5. Stop debugging and close Microsoft Visual Studio.

Results: After completing this exercise, you will be able to create a Photo Sharing application that displays well on mobile devices and devices with small screens.

Question: When you first browsed the web application in Exercise 1, why was the menu and the breadcrumb trail visible on the home page, but not on the All Photos page or any other page?

Question: When you first viewed the site as a mobile browser in Exercise 3, what are the problems you came across with the display of the site heading and menu?

Module Review and Takeaways

In this module, you learned how to apply a consistent look and feel to a web application, and share other common components, such as headers and footers, between all views. You also learned how to use the CSS and display modes to adapt the web application for smaller screens and mobile devices. You also familiarized yourself with HTML5 elements that allow you to develop web applications that work on various browsers and devices.

Real-world Issues and Scenarios

When you develop web applications, you need to create applications that work on different devices and browsers, such as iPhone, iPad, Windows Phone, Google Chrome, and Internet Explorer 10. In such cases, you can use the HTML5 elements and features in MVC 4, such as mobile-specific views, media queries, and jQuery Mobile library, to create applications that work well in various browsers and devices.

Review Question(s)

Question: You are building an application, which needs to work in different mobile devices, Windows Phone, Windows, and Mac. You want to reduce the effort for maintaining the code which is required for different devices and you want to ensure that it would work with new browsers. What should you do?

Module 09

Building Responsive Pages in ASP.NET MVC 4 Web Applications

Contents:

Module Overview

Many web applications need to display large amount of information and graphics. Large volumes of data make web applications take longer to load. Instead of having all the elements on the page load multiple times, you can implement caching in your web applications to reduce the need to repeatedly load the same elements. You can also use partial page updates to reduce the need to load the entire webpage, by enabling the application to refresh only specific sections of the webpage.

Objectives

After completing this module, you will be able to:

- Implement partial page updates that help reduce the need to reload entire pages.
- Implement caching to reduce the loading time of the different elements of a webpage.

Lesson 1
Using AJAX and Partial Page Updates

While developing web applications, to update individual sections in the page, you may need to reload the entire webpage. Asynchronous JavaScript and XML (AJAX) in ASP.NET MVC 4 enables partial page updates, to help update sections of a webpage, without reloading the entire page. The **Ajax.ActionLink** helper helps implement partial page updates in your web application.

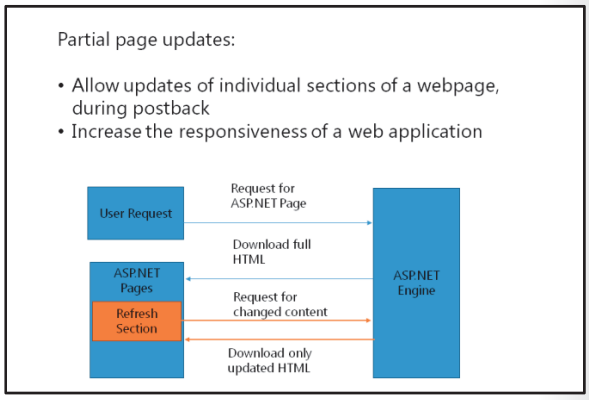
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of using partial page updates.
- Use AJAX in an ASP.NET MVC 4 web application.
- Use the **Ajax.ActionLink** helper.

Why Use Partial Page Updates?

ASP.NET and MVC facilitate server-side processing. Server-side processing enables HTML rendering to occur at the server side and the server usually generates the HTML. When you update a webpage, any updates or actions performed on the page require a round-trip request to the server. Such constant server requests affect the performance of the application.



The AJAX development model helps reduce the need for refreshing an entire webpage, each time an update of page content is required. AJAX uses JavaScript and XML to obtain information from the client system. AJAX creates webpages based on the XML information downloaded from the server. However, developing web applications by using AJAX is not easy, because it requires using complex technologies, such as JavaScript and XML. Microsoft includes a feature in ASP.NET called partial page updates that functions along with AJAX to reduce the need for refreshing an entire webpage, each time an update occurs.

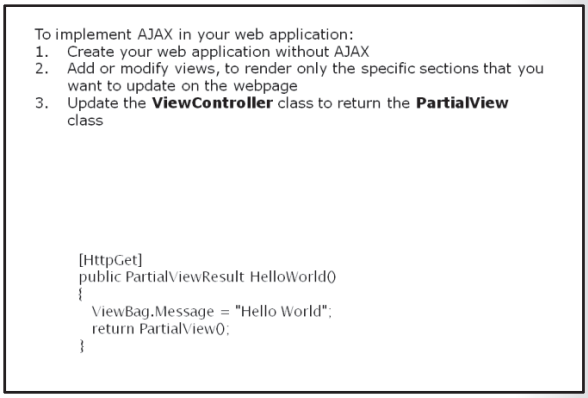
Partial page updates use AJAX technologies to help update individual sections of a webpage, during postback. Partial page updates:

- Require fewer lines of code.
- Help reduce the data sent to users, each time a webpage update occurs.
- Increase the responsiveness of the web application.

Question: How do partial page updates help in improving the responsiveness of a web application?

Using AJAX in an MVC 4 Web Application

To implement AJAX in your MVC 4 application, you need to create views that render only the updated content, and not the entire webpage. You can initially develop your web application without using AJAX, and then check the application for any functionality errors. This practice helps reduce the time required to troubleshoot the application. This practice also helps separate any application functionality errors from errors that occur while implementing AJAX.



To implement partial page updates, you need to create a view, called a partial view, which includes only the section that you need to update.

For example, consider a view as shown in the following code.

An Example View

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        <div id="divMessage">@ViewBag.Message</div>
         @Html.ActionLink("Refresh","HelloWorld")
    </div>
</body>
</html>
```

In the preceding code example, **ActionLink** helps direct users to another view called **HelloWorld**.

The following code shows the **HelloWorld** view.

The Hello World View

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>HelloWorld</title>
</head>
<body>
    <div>
        @ViewBag.Message
    </div>
</body>
</html>
```

To implement AJAX in the preceding code example, you need to update the **HelloWorld** view to render only the content that is updated.

Consider another example wherein you want to render only an updated message, on a webpage. The following code shows a view that is updated to render only the changed message.

A ViewBag Example

```
@ViewBag.Message
```

With AJAX, the Javascript retrieves only a specific portion of a webpage, which you want to update, from the server. In the **ViewController** class, you need to update the **View** function to return the **PartialView** class, instead of the entire **View** class.

The following code shows how to update the **View** function to return the **PartialView** class.

Returning a Partial View in a Controller Action

```
public class Default1Controller : Controller
    {
        //
        // GET: /Default1/
        public ActionResult Index()
        {
            ViewBag.Message = "Hello";
            return View();
        }
        [HttpGet]
        public PartialViewResult HelloWorld()
        {
            ViewBag.Message = "Hello World";
            return PartialView();
        }
    }
```

Optionally, you can add the **HttpGet** or **HttpPost** attributes before the **View** function. These attributes help indicate if the partial page update should be performed over the **HTTP POST** or **HTTP GET** method.

Question: What is the mandatory action that you should perform to implement partial page updates in your web application?

The Ajax.ActionLink Helper

You can use the **Ajax.ActionLink** helper to trigger partial page updates. The **Ajax.ActionLink** helper helps initiate the Javascript, to obtain the updated HTML information from the view and replace or insert the updated HTML information at a specific location.

The following code shows how to use the **Ajax.ActionLink** helper.

The **Ajax.ActionLink** helper:
Helps obtain updated HTML information from the view
Helps replace content in a specific location

```
@Ajax.ActionLink(
  "Refresh",
  "HelloWorld",
  new AjaxOptions{
    HttpMethod = "POST",
    UpdateTargetId = "divMessage",
    InsertionMode = InsertionMode.Replace
  }
)
```

Using the Ajax.ActionLink Helper

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
```

```
    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.3.js"
type="text/javascript"></script>
    <script src="http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.unobtrusive-ajax.min.js"
         type="text/javascript"></script>
</head>
<body>
    <div>
        <div id="divMessage">@ViewBag.Message</div>
         @Ajax.ActionLink("Refresh","HelloWorld", new AjaxOptions{ HttpMethod = "POST",
UpdateTargetId = "divMessage", InsertionMode = InsertionMode.Replace })
    </div>
</body>
</html>
```

In the preceding example, parameters such as **HttpMethod** and **UpdateTargetId** are included along with the **Ajax.ActionLink** helper, to:

- Obtain the HTML information from the **HelloWorld** view, by using the **HTTP POST** method.
- Replace the content in the **divMessage** HTML element.

While using the **Ajax.ActionLink** helper, you need to include the jQuery and jQuery unobtrusive libraries in the same webpage, because the **Ajax.ActionLink** helper uses scripts from these two libraries.

Question: What is the primary function of the **Ajax.ActionLink** helper?

Lesson 2
Implementing a Caching Strategy

Web applications display information on a webpage by retrieving the information from a database. If the information that should be retrieved from the database is large, the application may take longer to display the information on a webpage. ASP.NET MVC 4 supports some caching techniques to help reduce the time required to process a user request.

Before implementing caching, you should first analyze if caching is relevant to your application, because caching is irrelevant to webpages whose content change frequently. To successfully implement caching in your web application, you need to familiarize yourself with the various types of caches, such as output cache, data cache, and HTTP cache.

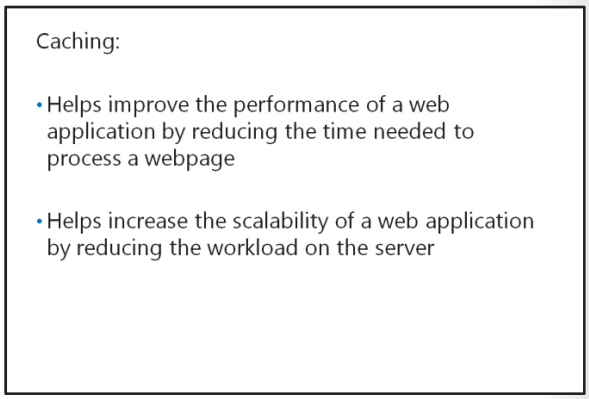
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of using caching.
- Describe the output cache.
- Describe the data cache.
- Describe the HTTP cache.
- Describe how to prevent caching for webpage content that changes frequently.
- Configure caching.

Why Use Caching?

Caching involves storing the information that is obtained from a database in the memory of a web server. If the content rendered by a webpage is static in nature, the content can be stored in caches or proxy servers. When a user requests content from a web application, caching ensures that the user receives content from the cache, thereby eliminating the need for repeated real-time processing.



Caching:

- Reduces the need to repeatedly retrieve the same information from the database.
- Reduces the need to reprocess data, if a user places a request multiple times.
- Helps improve the performance of a web application, by reducing the load on servers.
- Helps increase the number of users who can access the server farm.

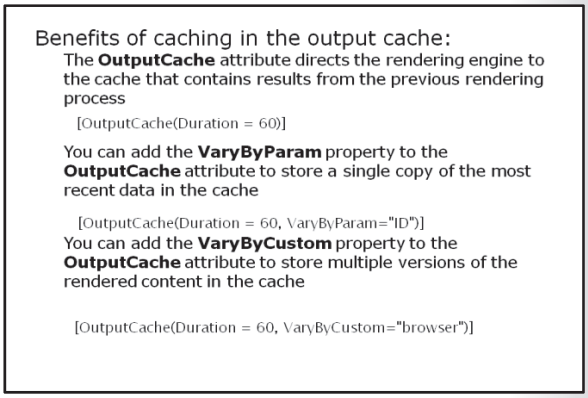
However, caching does not help web applications that include frequent content changes. This is because, the content rendered from a cache may be outdated, when compared to the current information. Therefore, you should evaluate the content of your web application and analyze the impact of rendering outdated content, before implementing caching.

Question: How does caching help increase the scalability of a web application?

The Output Cache

Output cache allows ASP.NET engines to store the rendered content of a webpage in the memory of the web server. Therefore, when a user requests a specific page multiple times, the content is retrieved from the cache, thereby avoiding the execution of programming code in the server.

Output cache is a good complement to AJAX partial page updates. Output cache and partial page updates reduce the workload on the server and increase the number of user requests that a server can handle.



In ASP.NET MVC 4, you can implement output caching, by adding the **OutputCache** attribute to the controller.

The following code shows how to implement output caching.

Configuring the Output Cache

```
 [OutputCache(Duration = 60)]
public PartialViewResult HelloWorld()
{
   ViewBag.Message = "Hello World";
   return PartialView();
}
```

The **OutputCache** attribute helps direct the rendering engine to the cache that contains results from the previous rendering process. The **Duration** parameter of the **OutputCache** attribute helps control the period of time in seconds for which data should be stored in the cache.

By default, the output cache stores only one copy of the rendered content, for each view. Consider a view with the **QueryString** input parameter that enables content to change based on the variable gathered from the database or a prior request. In this case, you can add the **VaryByParam** property to the **OutputCache** attribute, to store a copy of each unique combination of parameters in the cache.

The following code shows how to add the **VaryByParam** property to the **OutputCache** attribute.

Caching by Parameter Value

```
 [OutputCache(Duration = 60, VaryByParam="ID")]
public PartialViewResult HelloWorld()
{
    ViewBag.Message = "Hello World";
    return PartialView();
}
```

In the preceding example, observe that the **VaryByParam** property refers to **QueryString**, instead of other MVC parameters. You can also use the **VaryByCustom** property.

The following code shows how to add the **VaryByCustom** property to the **OutputCache** attribute.

Using VaryByCustom

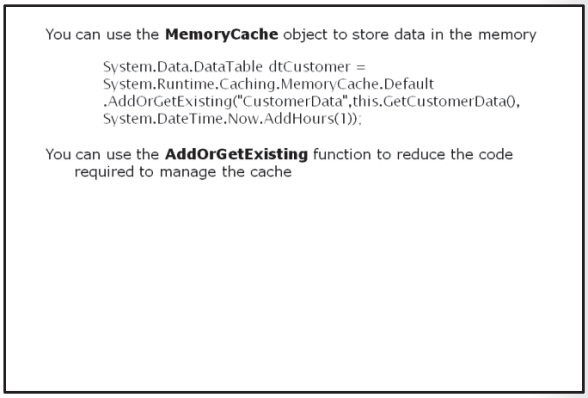
```
 [OutputCache(Duration = 60, VaryByCustom="browser")]
public PartialViewResult HelloWorld()
{
    ViewBag.Message = "Hello World";
    return PartialView();
```

```
}
```

You can add **browser** as the input parameter to the **VaryByCustom** property. The **browser** parameter helps store a copy of the rendered content corresponding to each browser that the application is run on. If you want to control and implement your own logic to determine when a new copy should be stored, you need to override the **GetVaryByCustomString** function in the **Global.asax** file of your project.

Question: How does the functioning of a web application that implements the output cache differ from an application that does not implement output cache?

The Data Cache



Web applications usually depend on the content in a database, to render content on a webpage. Databases sometimes encounter performance issues caused by poorly written queries, which can slow down the performance of database requests resulting in poor webpage performance. You can implement the data cache in your web application to avoid loading data from a database every time a user places a request. The **MemoryCache** class allows you to implement data cache in your web application. Implementing the data cache involves the following actions:

1. Loading information from the database
2. Storing content in the **MemoryCache** object
3. Retrieving data from the **MemoryCache** object
4. Ensuring that content is available in the **MemoryCache** object; otherwise, reloading the content

The following code shows how to add the **MemoryCache** object.

Using the Memory Cache

```
System.Data.DataTable dtCustomer = System.Runtime.Caching.MemoryCache.Default.
  AddOrGetExisting("CustomerData", this.GetCustomerData(),
  System.DateTime.Now.AddHours(1));
```

In the preceding example, the following parameters are specified:

- *Key*. The unique identifier of the object that should be stored in the memory cache.
- *Value*. The object that should be stored in the memory cache.
- *AbsoluteExpiration*. The time when the cache should expire.

You can use the **AddOrGetExisting** function, instead of the **Add** function, to enable the application to refresh and retrieve data in one line of code. If the cache contains the relevant data, the **AddOrGetExisting** function retrieves the data from the cache. If the cache does not contain the relevant data, the **AddOrGetExisting** function allows adding the data to the cache, and then rendering the same data on the webpage.

Question: What are the benefits of implementing data caching in MVC applications?

The HTTP Cache

You can implement HTTP caching in the Browser Cache and the Proxy Cache.

Browser Cache:

- Includes a copy of the web application stored in local computer drive
- Allows only one user to access data, at a time

Proxy Cache:

- Includes a copy of the web application stored on a centralized server
- Allows multiple users to access data, at a time

The Browser Cache

Most web browsers store the content downloaded from web servers in their local cache. Storing data in the local cache helps remove the need to repeatedly download content from the server. Web browsers frequently check content for updates. If the content is updated in the server, web browsers download the content from the server, to attend to user requests. Otherwise, web browsers render content from the local cache.

The Proxy Cache

The functionality of the proxy cache is similar to the functionality of the browser cache. However, the cache is stored on a centralized server. Users can connect to the Internet or web servers by using this proxy server. Proxy servers store a copy of a web application in a manner similar to a web browser storing a copy of an application in the local drives. Many users can access the cache in a proxy server, while only one user can access the browser cache at a time.

Question: What is the difference between HTTP cache and output cache?

Preventing Caching

Caching can sometimes create issues in web applications, because if an application involves frequent content updates, caching prevents users from viewing these content updates. To resolve caching issues, you can implement an HTTP header called Cache-Control. The Cache-Control header indicates to the web browser how to handle the local cache. All HTTP clients, such as browsers and proxy servers, respond to the instructions provided in the Cache-Control header to determine how to handle the local cache of a web application.

You can set the Cache-Control header value to **HttpCachePolicy.SetCacheability** to control the caching performance:

```
Response.Cache.SetCacheability(HttpCacheability.Private);
```

You can set the Cache-Control header value to **NoCache** to prevent the caching performance:

```
Response.Cache.SetCacheability(HttpCacheability.NoCache);
```

Additional Reading: For more information about setting this header, go to: http://go.microsoft.com/fwlink/?LinkID=288972&clcid=0x409

You can use the **HttpCachePolicy.SetCacheability** method to specify the value of the Cache-Control header. The **HttpCachePolicy.SetCacheability** method helps control caching performance.

The following code shows how to use the **HttpCachePolicy.SetCacheability** method.

Using SetCacheability

```
Response.Cache.SetCacheability(HttpCacheability.Private);
```

In the preceding example, the **HttpCachePolicy.SetCacheability** method takes the **Private** enumeration value.

To prevent caching in your web application, you should set the Cache-Control header value to **NoCache**.

The code shows how to exclude the HTTP cache in your web application.

Preventing Caching

```
Response.Cache.SetCacheability(HttpCacheability.NoCache);
```

Question: What scenarios would require you to prevent caching for a web application?

Demonstration: How to Configure Caching

In this demonstration, you will see:

- How to configure the output cache for an MVC controller action.
- Measure the difference that the configured cache makes to the delivery of the page.

Demonstration Steps

1. On the **DEBUG** menu of the **OperasWebSite – Microsoft Visual Studio** window, click **Start Debugging**.
2. On the Operas I Have Seen page, click the **Tools** button, and then click **F12 developer tools**.
3. On the **Cache** menu of the developer window, click **Always refresh from server**.
4. On the **Network** tab of the developer window, click **Start Capturing**.
5. On the Operas I Have Seen page, click the **All Operas** link.
6. When the page is fully loaded, in the developer window, click **Stop Capturing**.
7. In the URL section of the developer window, click **http://localhost:**<*portnumber*>**/Opera**, and then click **Go to detailed view**.
8. On the **Timings** tab, click the **Request** entry.
9. In the **Duration** column, note the value displayed.
10. On the **Network** tab, click **Clear**, and then click **Start capturing**.
11. On the Operas I Have Seen page, click the **All Operas** link.
12. When the page is fully loaded, in the developer window, click **Stop capturing**.
13. In the URL section of the developer window, click **http://localhost:**<*portnumber*>**/Opera**, and then click **Go to detailed view**.
14. On the **Timings** tab, click the **Request** entry.
15. In the **Duration** column, note the value displayed.

Note: The time taken by the server to render the **/Opera** page and return the page to the browser is similar to the time taken by the server in the first instance. The page is not cached.

16. In the Windows Internet Explorer window, click the **Close** button.
17. In the Solution Explorer pane of the **OperasWebSite – Microsoft Visual Studio** window, under OperasWebSite, expand **Controllers**, and then click **OperaController.cs**.
18. In the OperaController.cs code window, locate the following code.

```
using System.Web.Mvc;
```

19. Place the mouse cursor at the end of the located code, press Enter, and then type the following code.

```
using System.Web.UI;
```

20. In the OperaController.cs code window, locate the following code.

```
public ActionResult Index()
```

21. Place the mouse cursor immediately before the located code, press Enter, and then type the following code.

```
[OutputCache(Duration=600, Location=OutputCacheLocation.Server, VaryByParam="none")]
```

22. On the **DEBUG** menu of the **OperasWebSite – Microsoft Visual Studio** window, click **Start Debugging**.
23. On the **Cache** menu of the developer window, click **Always refresh from server**.
24. On the **Network** tab, click **Start capturing**.
25. On the Operas I Have Seen page, click the **All Operas** link.
26. When the page is fully loaded, in the developer window, click **Stop capturing**.
27. In the URL section of the developer window, click **http://localhost:**<*portnumber*>**/Opera**, and then click **Go to detailed view**.
28. On the **Timings** tab, click the **Request** entry.
29. In the **Duration** column, note the value displayed.
30. On the **Network** tab, click **Clear**, and then click **Start capturing**.
31. On the Operas I Have Seen page, click the **All Operas** link.
32. When the page is fully loaded, in the developer window, click **Stop capturing**.
33. In the URL section of the developer window, click **http://localhost:**<*portnumber*>**/Opera**, and then click **Go to detailed view**.
34. On the **Timings** tab, click the **Request** entry.
35. In the **Duration** column, note the value displayed.

Note: Note that the time taken by the server to render the **/Opera** page and return the page to the browser is significantly less than the time taken by the server in the first instance.

36. On the **File** menu of the developer window, click **Exit**.
37. In the Windows Internet Explorer window, click the **Close** button.
38. In the **OperasWebSite – Microsoft Visual Studio** window, click the **Close** button.

Lab: Building Responsive Pages in ASP.NET MVC 4 Web Applications

Scenario

Your manager has asked you to include comments for photos in the Photo Sharing application. Your manager has also highlighted that the performance of some pages in the application is too slow for a production site.

You want to ensure that comments for photos take minimal loading time, for which you decide to use partial page updates. You also want to return pages in quick time, while updated information is displayed, for which you decide to configure caching in your application.

Objectives

After completing this lab, you will be able to:

- Write controller actions that can be called asynchronously and return partial views.
- Use common AJAX helpers to call asynchronous controller actions, and insert the results into Razor views.
- Configure ASP.NET caches to serve pages in quick time.

Lab Setup

Estimated Time: 60 minutes

Virtual Machine: **20486B-SEA-DEV11**

Username: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

a. On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.

b. In the navigation pane of the **Options** dialog box, click **Package Manager**.

c. Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Using Partial Page Updates

Scenario

You have been asked to include a comment functionality on the photo display view of the Photo Sharing application. You want to ensure high performance by using AJAX partial page updates.

In this exercise, you will

- Import a partially complete controller to add comments, and a view to delete comments.

Add code to the controller for partial page update.

The main tasks for this exercise are as follows:

1. Import the Comment controller and Delete view.

2. Add the _CommentsForPhoto action and view.

3. Add the _Create Action and the _CreateAComment views.

4. Add the _CommentsForPhoto POST action.

5. Complete the _CommentsForPhoto view.

▶ Task 1: Import the Comment controller and Delete view.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual Machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **PhotoSharingApplication.sln** file from the following location:
   - File location: Allfiles (D):\Labfiles\Mod09\Starter \PhotoSharingApplication
3. Create a new folder in the **Views** folder by using the following information:
   - Name of the new folder: **Comment**
4. Add an existing item to the new **Comment** folder by using the following information:
   - File location of the existing item: **Allfiles (D):\Labfiles\Mod09\Comment Components\Delete.cshtml**
5. Add an existing item to the **Controller** folder by using the following information:
   - File location of the existing item: **Allfiles (D):\Labfiles\Mod09\Comment Components\CommentController.cs**

▶ Task 2: Add the _CommentsForPhoto action and view.

1. Add a new action to **CommentController.cs**. by using the following information:
   - Annotation: **ChildActionOnly**
   - Scope: **public**
   - Return type: **PartialViewResult**
   - Name: **_CommentsForPhoto**
2. Parameter: an integer named **PhotoId**
3. In the **_CommentsForPhoto** action, select all the comments in the database that have a **PhotoID** value equal to the **PhotoId** parameter, by using a LINQ query.
4. Save the **PhotoId** parameter value in the **ViewBag** collection to use it later in the view.
5. Return a partial view as the result of the **_CommentsForPhoto** action by using the following information:
   - View name: **_CommentsForPhoto**
   - Model: **comments.ToList()**
6. Add a new partial view to display a list of comments by using the following information:
   - Parent folder: **Views/Shared**
   - View name: **_CommentsForPhoto**
   - View type: **Strong**.
   - Model class: **Comment**

- Create partial view: **Yes**.

7. Bind the **_CommentsForPhoto.cshtml** view to an enumerable collection of comments.
8. Create an **H3** element by using the following information:
   - Heading: **Comments**
9. After the heading, create a **DIV** element with the ID **comments-tool**. Within this **DIV** element, create a second **DIV** element with the ID **all-comments**.
10. For each item in the model, render a **DIV** element with the **photo-comment** class.
11. Within the ** ** element, add a **DIV** element with the **photo-comment-from** class. Within this **DIV** element, render the **UserName** value of the model item by using the **Html.DisplayFor()** helper.
12. Add a **DIV** element with the **photo-comment-subject** class. Within this **DIV** element, render the **Subject** value of the model item by using the **Html.DisplayFor()** helper.
13. Add a **DIV** element with the **photo-comment-body** class. Within this **DIV** element, render the **Body** value of the model item by using the **Html.DisplayFor()** helper.
14. Render a link to the **Delete** action by using the **Html.ActionLink()** helper. Pass the **item.CommentID** value as the **id** parameter.
15. In the **Views/Photo/Display.cshtml** view file, just before the **Back To List** link, render the **_CommentsForPhoto** partial view by using the following information:
    - Helper: **Html.Action()**
    - Action: **_CommentsForPhoto**
    - Controller: **Comment**
16. PhotoId parameter: **Model.PhotoID**
17. Run the application in debugging mode and browse to **Sample Photo 1**. Observe the display of comments on the page.
18. Close Internet Explorer.

▶ Task 3: Add the _Create Action and the _CreateAComment views.

1. Add a new action to the **CommentController.cs** file by using the following information:
   - Scope: **public**
   - Return type: **PartialViewResult**
   - Name: **_Create**
2. Parameter: an integer named **PhotoId**.
3. In the **_Create** action, create a new **Comment** object and set its **PhotoID** property to equal the **PhotoId** parameter.
4. Save the **PhotoId** parameter value in the **ViewBag** collection to use it later in the view.
5. Return a partial view named **_CreateAComment**.
6. Add a new partial view for creating new comments by using the following information:
   - Parent folder: **Views/Shared**
   - View name: **_CreateAComment**
   - View type: **Strong**

- Model class: **Comment**
- Create partial view: **Yes**

7. In the **_CreateAComment** view, render validation messages by using the **Html.ValidationSummary()** helper. For the **excludePropertyErrors** parameter, pass **true**.
8. After the validation messages, add a **DIV** element with the **add-comment-tool** class.
9. Within the ** ** element, add a **DIV** element with no class or ID.
10. Within the **DIV** element you just created, add a **SPAN** element with the **editor-label** class and content **Subject:**
11. After the **SPAN** element you just created, add a second **SPAN** element with the **editor-field** class. Within this element, render the **Subject** property of the model by using the **Html.EditorFor()** helper.
12. Within the ** ** element, add a second **DIV** element with no class or ID.
13. Within the **DIV** element you just created, add a **SPAN** element with the **editor-label** class and content **Body:**
14. After the **SPAN** element you just created, add a second **SPAN** element with the **editor-field** class. Within this element, render the **Body** property of the model by using the **Html.EditorFor()** helper.
15. Within the ** ** element, add an **INPUT** element by using the following information:
    - Element: **<input>**
    - Type: **submit**
16. Value: **Create**
17. Save all your changes.

▶ Task 4: Add the _CommentsForPhoto POST action.

1. Add a new action to the **CommentController.cs** file by using the following information:
    - Annotation: **HttpPost**
    - Scope: **public**
    - Return type: **PartialViewResult**
    - Name: **_CommentsForPhoto**
    - Parameter: a **Comment** object named **comment**.
2. Parameter: an integer named **PhotoId**.
3. In the **_ComentForPhoto** action, add the **comment** object to the **context** and save the changes to the **context**.
4. Select all the comments in the database that have a **PhotoID** value equal to the **PhotoId** parameter by using a LINQ query.
5. Save the **PhotoId** parameter value in the **ViewBag** collection to use it later in the view.
6. Return a partial view as the result of the **_CommentsForPhoto** action by using the following information:
    - View name: **_CommentsForPhoto**
7. Model: **comments.ToList()**

8. Save all the changes.

▶ Task 5: Complete the _CommentsForPhoto view.

1. In the **_CommentsForPhoto.cshtml** view file, use a **using{}** block to render an HTML form around all tags by using the following information:
   - Helper: **Ajax.BeginForm()**
   - Action name: **_CommentsForPhoto**
   - PhotoId parameter: **ViewBag.PhotoId**
2. Ajax options: **UpdateTargetId = "comment-tool"**
3. In the form code block, in the ** ** element, add a new **DIV** element with the **add-comment-box** class and the ID **add-comment**.
4. In the **DIV** element you just created, render the **_Create** action of the **Comment** controller by using the **Html.Action()** helper. Pass the **ViewBag.PhotoId** value as the **PhotoId** parameter.
5. Add script tags to the **_MainLayout.cshtml** page that reference the following content delivery network (CDN) locations:
   - http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.8.0.min.js
6. http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.unobtrusive-ajax.js
7. Start the web application in debugging mode, browse to **Sample Photo 1**, and observe the comments displayed.
8. Add a new comment to Sample Photo 1.
   - Subject: **Test Comment**
9. Body content: **This comment is to test AJAX-based partial page updates.**
10. Stop debugging.

Results: At the end of this exercise, you will have ensured that new comments can be added and displayed on the pages of the application without a complete page reload. You will create a Photo Sharing application with a comments tool, implemented by using partial page updates.

Exercise 2: Optional—Configuring the ASP.NET Caches

Scenario

You have been asked to configure the ASP.NET caches in the Photo Sharing application to ensure optimal performance. Senior developers are particularly concerned that the All Photos gallery might render slowly because it will fetch and display many photos from the database at a time.

In this exercise, you will:

- Configure the output cache to store the photo index view.
- Use the developer tools in Internet Explorer to examine the speed at which image files and pages render with and without caching.
- Configure the output cache to store the results of the GetImage action so that image files can be returned from the cache.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Test the All Photos page with no caching.
2. Configure caching for the Index action.
3. Retest the All Photos page with Index caching.
4. Configure caching for the GetImage action.
5. Retest the All Photo page with GetImage caching.

▶ Task 1: Test the All Photos page with no caching.

1. Start the application in debugging mode and configure the browser to always refresh the page from the server by using the Internet Explorer developer tools.
2. Capture traffic between the browser and the server when the **All Photos** page is loaded, by using the Network tools.
3. Record the time taken by the server to render the **/Photo** page and return the page to the browser. This value is the **Request** duration, which you can find on the **Timings** tab.
4. Clear the first network capture, and capture a second request to the **All Photos** page.
5. Record the second instance of time taken by the server to render the **/Photo** page and return the page to the browser. Observe if the duration is more or less than the first instance.
6. Stop debugging.

▶ Task 2: Configure caching for the Index action.

1. Open the **PhotoController.cs** code file, and add a **using** statement for the following namespace:
2. System.Web.UI
3. Configure the **Index** action to use the output cache by using the following information:
   - Duration: **10 minutes**
   - Location: **Server**
   - Vary by parameters: **None**
4. Save all your changes.

▶ Task 3: Retest the All Photos page with Index caching.

1. Start the application in debugging mode, and configure the browser to always refresh the page from the server, by using the Internet Explorer developer tools.
2. Capture the traffic between the browser and the server when the **All Photos** page is loaded, by using the Network tools.
3. Record the time taken by the server to render the **/Photo** page and return the page to the browser. This value is the **Request** duration, which you can find on the **Timings** tab.
4. Clear the first network capture, and capture a second request to the **All Photos** page.
5. Record the second instance of the time taken by the server to render the **/Photo** page and return the page to the browser. Observe if the duration is more or less than the first instance.
6. Record the time taken by the server to render the **/Photo/GetImage/1** request.
7. Stop debugging.

▶ Task 4: Configure caching for the GetImage action.

1. In the **PhotoController**, configure the **GetImage** action to use the output cache, by using the following information:
   - Duration: **10 minutes**.
   - Location: **Server**
2. Vary by parameters: **id**
3. Save all your changes.

▶ Task 5: Retest the All Photo page with GetImage caching.

1. Start the application in debugging mode and configure the browser to always refresh the page from the server, by using the Internet Explorer developer tools.
2. Capture the traffic between the browser and the server when the **All Photos** page is loaded, by using the Network tools.
3. Record the time taken by the server to render the **/Photo/GetImage/1** request.
4. Clear the first network capture, and capture a second request to the **All Photos** page.
5. Record the second instance of the time taken by the server to render the **/Photo/GetImage/1** request and return the page to the browser.
6. Close the developer tools, stop debugging, and close Visual Studio.

Results: At the end of this exercise, you will create a Photo Sharing application with the Output Cache configured for caching photos.

Question: In Exercise 2, why was the Request timing for /Photo not reduced for the first request when you configured the output cache for the index action?

Question: In Exercise 2, when you configured the output cache for the GetImage() action, why was it necessary to set VaryByParam="id"?

Module Review and Takeaways

In this module, you used AJAX and partial page updates in MVC applications. AJAX and partial page updates help reduce the need for reloading the entire page, when a user places a request. Partial page updates also reduce the need for writing multiple lines of code, to update specific portions of a webpage. You also used caching to increase the performance of a web application.

Real-world Issues and Scenarios

Web applications usually run multiple queries to retrieve information from a database and render content on the webpages. Users sometimes complain that webpages take longer to load. Therefore, developers implement caching in the web application, to reduce the need to load data from a database, every time a user places a request. Caching helps webpages load faster, thereby increasing the performance of the application.

Review Question(s)

Question: An application is refreshing the content every 10 seconds for the updated information from database. User complaints that this is impacting their work and has caused data loss. How would you propose to help resolve this issue?

Module 10

Using JavaScript and jQuery for Responsive MVC 4 Web Applications

Contents:

Module Overview

Responsive web design is a web designing approach that helps create visually rich and interactive web applications. JavaScript plays an important role in the development of responsive web applications. You need to know how to use JavaScript to implement application logic and resize interface elements, without triggering a full-page refresh. To simplify adding JavaScript to your web application, you need to know how to use libraries such as jQuery, jQuery UI, and jQuery Mobile.

Objectives

After completing this module, you will be able to:

- Add JavaScript code to your web application.
- Use the jQuery and jQuery UI libraries, in your web application.

Lesson 1
Rendering and Executing JavaScript Code

You can create interactive HTML elements in your web application by using JavaScript. ASP.NET renders these interactive elements on your webpages. You can add packaged JavaScript libraries to your project by using the NuGet tool. You should know how to use AJAX to update the contents of webpages. By using AJAX, you can optimize the performance of your web application. In addition, you should know how the content delivery network (CDN) helps take content geographically closer to users.

Lesson Objectives

After completing this lesson, you will be able to:

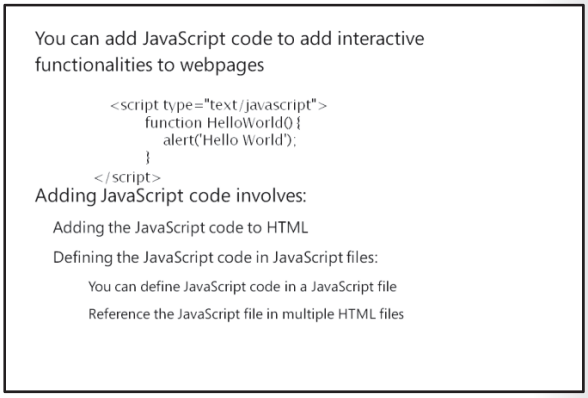
- Describe how to add JavaScript files to an MVC application.
- Describe how to call JavaScript functions in JavaScript libraries.
- Describe how to use JavaScript Libraries in MVC 4 web applications.
- List the benefits of using CDN to improve the performance of JavaScript libraries.
- Describe how to use the NuGet tool to add packages.
- Use the NuGet tool to add a JavaScript Library.

Adding JavaScript Files

You can add JavaScript code to web applications by:

- Adding the JavaScript code to HTML.
- Defining the JavaScript code in dedicated JavaScript files.

JavaScript code helps add interactive functionalities to the webpages of your application. The following example shows how to add JavaScript to HTML.



Inserting a JavaScript Function

```
<body>
    <script type="text/javascript">
        function HelloWorld() {
            alert('Hello World');
        }
    </script>
    <div>
       ...
    </div>
</body>
```

If you have multiple HTML pages in a web application, you need to add JavaScript code for each HTML page. You cannot simultaneously add JavaScript code for multiple HTML pages. Therefore, you can define the JavaScript code in a JavaScript file (.js file). Then, you can reference the JavaScript file in multiple HTML pages. This enables you to maintain a single JavaScript file, to edit the JavaScript code for multiple HTML pages. You can also have multiple JavaScript code files for a single HTML page.

The following image displays the Add New Item dialog box that helps to add a JavaScript file.

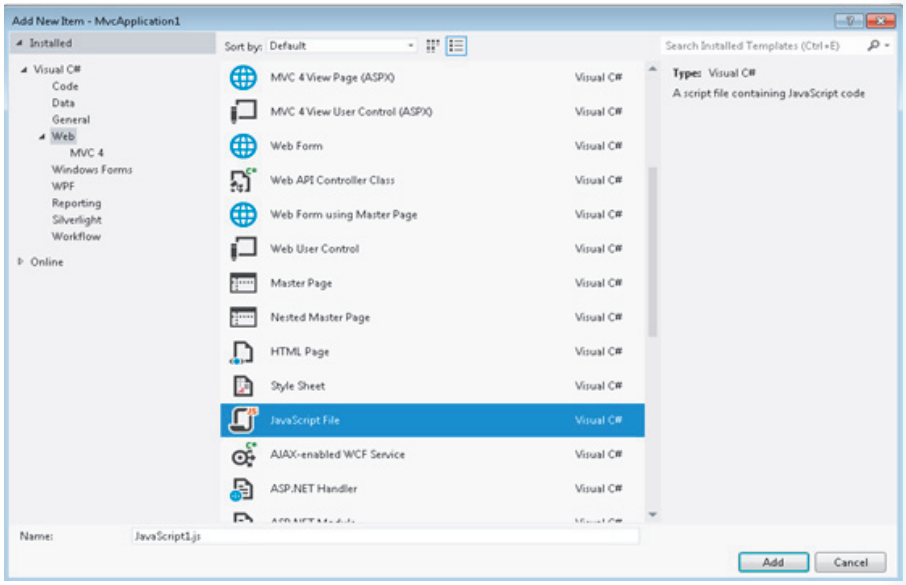


FIGURE 10.1:ADDING A JAVASCRIPT FILE

The following example shows how to reference the new JavaScript file in HTML pages.

Referencing JavaScript Files

```
<script src="~/Scripts/JavaScript1.js" type="text/javascript"></script>
```

You can create the Scripts folder in the root folder of your MVC project, and then save all JavaScript files in the Scripts folder.

Question: What are the advantages of using a JavaScript file?

Calling JavaScript Procedures

You can call functions defined in JavaScript files by using script blocks or event handlers.

The following code shows how to call the **HelloWorld** function from a script block.

Using a Script Block

```
<body>
    <script type="text/javascript">
        HelloWorld();
    </script>
    <div>
      Hello
    </div>
</body>
```

You can call JavaScript functions by using script blocks:

Define the JavaScript function in a script block

Reference the JavaScript file in HTML pages

```
<script type="text/javascript">
    HelloWorld()
</script>
```

You can also use events to trigger JavaScript functions:

Use the **onclick** event to initiate the JavaScript function assigned to an HTML file

```
<input type="button" value="Hello" onclick="HelloWorld();" />
```

Before calling a JavaScript function, you need to define the function by using a script block. Then, you must reference the JavaScript file from the HTML pages.

If you want to avoid calling the JavaScript function directly, you can use the **onclick** JavaScript event to trigger JavaScript functions. The **onclick** event initiates the JavaScript function assigned to an HTML file,

when you click the corresponding HTML element. JavaScript functions that are attached to document object model (DOM) events are called event handlers.

The following code shows how to add the HelloWorld event handler to the button's **onclick** event.

Using an Event Handler

```
<body>
    <div>
      Hello
        <input type="button" value="Hello" onclick="HelloWorld();" />
    </div>
</body>
```

Additional Reading: For more information about events that are available for HTML elements, go to http://go.microsoft.com/fwlink/?LinkID=288973&clcid=0x409

Question: What is the advantage of initiating JavaScript functions by using JavaScript events?

JavaScript Libraries

You can reduce the time taken to develop applications by using JavaScript libraries. JavaScript libraries help to:

- Reduce the amount of code you need to write to add functions.
- Reduce the time the system takes to debug the application.

Some commonly used JavaScript libraries include the following:

- jQuery (http://go.microsoft.com/fwlink/?LinkID=288974&clcid=0x409)
- jQuery UI (http://go.microsoft.com/fwlink/?LinkID=288975&clcid=0x410)
- jQuery Mobile (http://go.microsoft.com/fwlink/?LinkID=288976&clcid=0x411)
- jQuery Validation (http://go.microsoft.com/fwlink/?LinkID=288977&clcid=0x412)
- jQuery Cycle (http://go.microsoft.com/fwlink/?LinkID=288978&clcid=0x413)
- jQuery DataTables (http://go.microsoft.com/fwlink/?LinkID=288979&clcid=0x409)
- Prototype (http://go.microsoft.com/fwlink/?LinkID=299651&clcid=0x409)
- MooTools (http://go.microsoft.com/fwlink/?LinkID=299652&clcid=0x409)

JavaScript libraries:
- Help reduce the amount of code you need to write
- Help reduce the time taken to debug an application
- Help make web applications more interactive

Some commonly used JavaScript libraries include:
- jQuery
- jQuery UI
- jQuery Mobile
- jQuery Validation
- jQuery Cycle
- jQuery DataTables

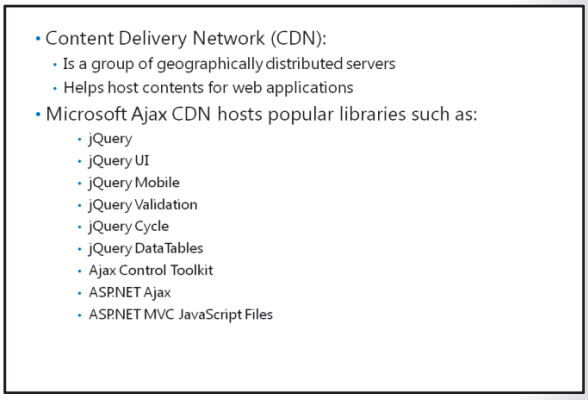
You can use JavaScript libraries to make your application more interactive. The functioning of JavaScript codes depends on the version of the library you use. Not all code may work with all versions of a library.

The jQuery library (and its related libraries) has the additional advantage of dealing with the differences in the DOM across different browsers and different browser versions.

Question: What is the advantage of using JavaScript libraries?

Using Content Delivery Networks for JavaScript Libraries

A content delivery network (CDN) is a group of geographically distributed servers used for hosting contents for web applications. In many cases, you can bring web content geographically closer to your applications users by using a CDN to host libraries. This will also improve the scalability and robustness of the delivery of that content.



The amount of content stored in a CDN varies among different web applications. Some applications store all their content on a CDN, while other applications store only some of their content.

Microsoft has a dedicated CDN called Microsoft Ajax CDN that hosts some popular JavaScript libraries, such as:

- jQuery
- jQuery UI
- jQuery Mobile
- jQuery Validation
- jQuery Cycle
- jQuery DataTables
- Ajax Control Toolkit
- ASP.NET Ajax
- ASP.NET MVC JavaScript Files

Note that Microsoft does not own the license of the JavaScript libraries mentioned in the preceding list. Microsoft only hosts the libraries for developers.

You can often reduce the loading time of your web application, by using the JavaScript libraries hosted on Microsoft Ajax CDN. Web browsers can cache these JavaScript libraries on a local system.

The following code shows how to use JavaScript libraries.

Linking to JavaScript Libraries

```
<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.8.0.js"
type="text/javascript"></script>
```

Additional Reading: For more information about Microsoft AJAX CDN, go to http://go.microsoft.com/fwlink/?LinkID=293689&clcid=0x409

Question: How can CDN help improve the performance of a web application?

Using the NuGet Tool to Add Packages

You can use the NuGet package manager to manage JavaScript libraries. You can avoid adding JavaScript libraries manually to your web application by installing the NuGet package manager in your application. This practice helps reduce the need for configuration tasks, while adding JavaScript libraries to an application.

Microsoft Visual Studio 2012 supports installing and using NuGet packages. You can search for NuGet packages in the NuGet store of Microsoft Visual Studio 2012. Then, you can directly install them into your MVC application.

- NuGet packages help add JavaScript libraries to your web application
- While using Microsoft Visual Studio 2012, you can:
  - Search for a NuGet package in the Nuget Store
  - Select the package that you want to use
  - On the Manage Nuget Packages page, click **Install**

The following image shows the application page that you can use to manage NuGet packages.

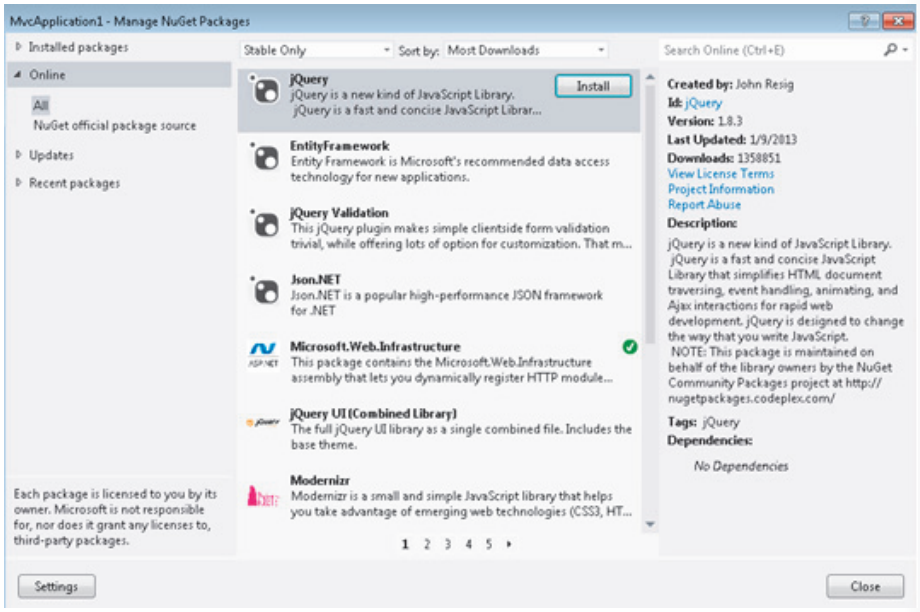


FIGURE 10.2:MANAGING NUGET PACKAGES

After you select the NuGet package that you wish to install, click **Install**, to download and install the package into your project.

Additional Reading: To search for NuGet packages and analyze the details of each package, go to http://go.microsoft.com/fwlink/?LinkID=288981&clcid=0x410

Question: Why should you use NuGet packages to add JavaScript libraries to your web application?

Demonstration: How to Use NuGet to Add a JavaScript Library

In this demonstration, you will see how to:

- Add the jQueryUI library to an application by using NuGet Package Manager.
- Access the location where jQueryUI components are added in the MVC application.

- Link to a script file in the site template view.

Demonstration Steps

1. In the Solution Explorer pane of the

OperasWebSite - Microsoft Visual Studio

window, expand **OperasWebSite**.

Note: There is no folder named Scripts at the top level of the project.

2. In the Solution Explorer pane, expand **Content**.

Note: The Content folder has only one file named **OperaStyles.css**, and there are no sub-folders.

3. On the **PROJECT** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Manage NuGet Packages**.
4. In the navigation pane of the **OperasWebSite – Manage NuGet Packages** dialog box, ensure **Online** is selected, and then click **All**.
5. In the result pane of the **OperasWebSite – Manage NuGet Packages** dialog box, click **jQuery UI(Combined Library)**, click **Install**, and then, when the installation is complete, click **Close**.
6. In the Solution Explorer pane, expand **Scripts**.

Note: NuGet Package Manager has added five files for jquery and jqueryUI to the application. Note the version number for jquery and jqueryUI.

7. In the Solution Explorer pane, under Contents, expand **themes**, expand **base**, and then click **jquery-ui.css**.

Note: NuGet Package Manager has added style sheets to the Content folder. These styles are used to set the styles for jQueryUI widgets, and the most important of these style sheets is **jquery-ui.css**.

8. In the Solution Explorer pane, collapse **base**, expand **Views**, and then expand **Shared**.
9. In the Solution Explorer pane, under Shared, click **_SiteTemplate.cshtml**.
10. In the _SiteTemplate.cshtml code window, locate the following code.

```
</head>
```

11. Place the mouse cursor before the located code, type the following code, and then press Enter.

```
<script type="text/javascript" src="@Url.Content("~/Scripts/jquery-ui-
1.10.0.js")"></scripts>
```

Note: In the above code, note that the version number provided is 1.10.0. When you type the code, replace the version number 1.10.0 with the latest version number.

12. In the _SiteTemplate.cshtml code window, locate the following code.

```
<title>@ViewBag.Title</title>
```

13. Place the mouse cursor at the end of the located code, press Enter, and then type the following code.

```
<link type="text/css" rel="stylesheet"
href="@Url.Content("~/Content/themes/base/jquery-ui.css")" />
```

Note: You can now use jQueryUI calls on any views in the application.

14. On the **FILE** menu of the **OperasWebSite - Microsoft Visual Studio** window, click **Save All**.
15. In the **OperasWebSite - Microsoft Visual Studio** window, click the **Close** button.

Lesson 2
Using jQuery and jQueryUI

jQuery is a JavaScript library that simplifies the adding of JavaScript to HTML pages. jQuery is an open-source software that you can use for free. It helps reduce the amount of code that you need to write, to perform tasks such as accessing and modifying HTML elements on a webpage. You can use the **ajax** function in jQuery to call a web service in your application. You can also use jQuery UI to add interactions, animations, effects, and widgets to your web applications.

Lesson Objectives

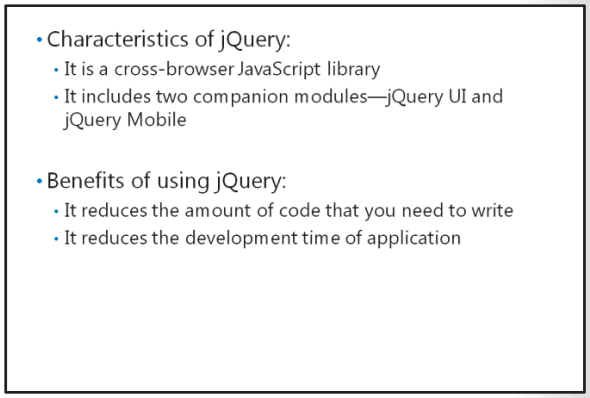
After completing this lesson, you will be able to:

- Describe jQuery.
- Describe how to link web applications to jQuery Libraries for client-side scripting.
- Describe how jQuery helps access the HTML elements of a webpage.
- Describe how to modify elements by using jQuery.
- Describe how to call a web service by using jQuery.
- Describe jQuery UI.
- Add a jQuery UI widget to an MVC 4 web application.

Introduction to jQuery

jQuery is a JavaScript library that you can use with different browsers. jQuery was first released in 2006. jQuery helps query the HTML Document Object Model (DOM) and obtain a set of HTML DOM elements. This feature of jQuery helps:

- Reduce the amount of code that you need to write, to perform a task.
- Reduce the development time of HTML applications.



jQuery includes the following features:

- DOM element selections
- DOM traversal and modification
- DOM manipulation, based on CSS selectors
- Events
- Effects and animations
- AJAX
- Extensibility through plug-ins
- Utilities
- Compatibility methods
- Multi-browser support

Additional Reading: For more information about jQuery, go to: http://go.microsoft.com/fwlink/?LinkID=288982&clcid=0x411

The jQuery family includes the following two companion modules:

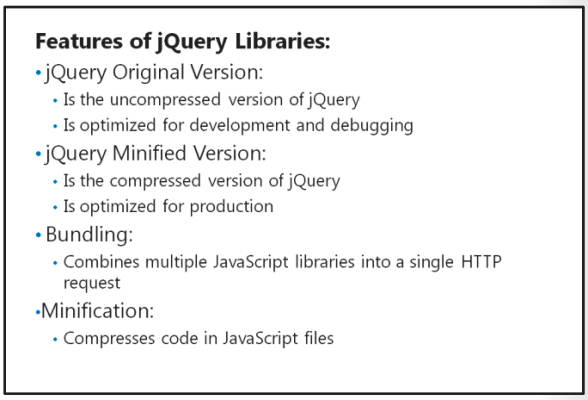
- *jQuery UI*. This library adds functions and other supporting elements that help implement a rich interface to HTML-based applications.
- *jQuery mobile*. This library adds functions that optimize the application interface for mobile devices.

Question: Why should you use jQuery while developing web applications?

Linking to jQuery Libraries

jQuery Original Version and jQuery Minified Version provide similar functionalities; however, they optimize web applications for different purposes:

- *jQuery Original Version* (jQuery-<*version*>.js). This is the uncompressed version of the jQuery library, with files sizes larger than 200 kilobytes (KB).
- *jQuery Minified Version* (jQuery-<*version*>.min.js). This includes the compressed and gZip versions of jQuery, with file sizes of about 30 KB.



When you develop the production environment, you can use jQuery Minified Version to reduce the loading time of the web application. If you use the minified version while working on the development environment, you cannot access the source code of the JavaScript libraries during the debug operation. Therefore, you can use the original version of jQuery, while creating the development environment.

The following code shows how to reference the jQuery library in your web application. You can add the line of code in the **<head>** element of your HTML.

Referencing jQuery

```
<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.8.0.min.js"
type="text/javascript"></script>
```

Bundling and Minification

Bundling is a new feature in ASP.NET 4.5 that you can use in MVC 4 web applications. Bundling helps combine multiple JavaScript libraries into a single HTTP request.

Minification compresses code files before incorporating them in the client application. Bundling and minification help reduce the loading time of web applications by reducing both the number and size of HTTP requests.

To use bundling and minification in your web application, you should perform the following steps:

1. Reference the **Microsoft.AspNet.Web.Optimization** library in your application by using NuGet packages.
2. In the App_Start folder of your project, add the BundleConfig.cs file.

3. In the BundleConfig.cs file, add the following code.

```
public static void RegisterBundles(System.Web.Optimization.BundleCollection bundles)
        {
            bundles.Add(new
System.Web.Optimization.ScriptBundle("~/bundles/jquery").Include(
                        "~/Scripts/jquery-{version}.js","~/Scripts/JavaScript1.js"));
        }
```

4. In the **Global.asax Application_Start** event, add the following code.

```
BundleConfig.RegisterBundles(System.Web.Optimization.BundleTable.Bundles);
```

The **ScriptBundle** class enables you to define the bundle that helps combine multiple JavaScript libraries into a single file. You can use a special placeholder, such as **{version}** in the JavaScript path, to help update the version of jQuery libraries. ASP.NET replaces **{version}** with the latest version number of the JavaScript libraries present in the Scripts folder. If the minified version of jQuery is available in the Scripts folder, the MVC 4 engine selects the minified version for bundling. Then, in your View file, you can include the following lines of code to render the bundled JavaScript file.

```
<head>
    @Scripts.Render("~/bundles/jquery")
</head>
```

You should add the following line of code in a page or namespace of the Web.config file in the View folder. This code helps trigger the functioning of **@Scripts.Render**.

```
<add namespace="System.Web.Optimization"/>
```

Based on the compilation setting in the Web.config file, the jQuery library renders minified versions of Javascript. For example, the following code sets the **debug** attribute of the **compilation** element to **false**, which allows jQuery to use the non-minified version of the libraries, making debugging of the libraries easier.

```
<compilation debug="false" />
```

Question: What are the benefits of using the minified version of jQuery in the production environment?

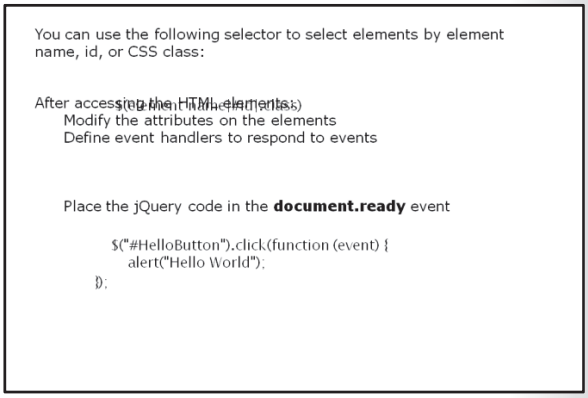
Accessing HTML Elements by Using jQuery

jQuery helps access HTML elements, to create interactive web applications. You use the following selector to select elements by element name, id, or CSS class, while adding jQuery code to access HTML elements.

```
$(element name|#id|.class)
```

The following jQuery selector accesses the HTML element with the **HelloButton** ID.

```
$("#HelloButton")
```



You can use jQuery to access or modify all instances of a specific HTML element, in an HTML page. The following jQuery selector identifies all instances of the **A** element in an HTML page.

```
$("a")
```

After accessing the HTML elements, you can perform actions on the elements, such as:

- Modifying the attributes on the HTML elements.
- Defining event handlers to respond to events associated with the selected HTML elements.

The following example adds an event handler to the click event of the **HelloButton** HTML element.

Using a jQuery Event Handler

```
$("#HelloButton").click(function (event) {
   alert("Hello World");
});
```

If the jQuery scripts load before the webpage loads, you may encounter errors such as **object not defined**. You can place the jQuery code in the **document.ready** event, to prevent the code from loading until all HTML elements in the page load.

The following code shows how to use the **document.ready** function

Using the Document Ready Function

```
$(document).ready(function () {
   //Code placed here will not execute before the page is fully loaded.
});
```

The following complete example shows how to access HTML elements by using jQuery.

A Complete Example

```
<body>
    <div>
      Hello
        <input type="button" value="Hello" id="HelloButton" />
    </div>
    <script type="text/javascript">
        $(document).ready(function () {
            $("#HelloButton").click(function (event) {
                alert("Hello World");
            });
        });
    </script>
</body>
```

Question: Why should you include the jQuery code in the **document.ready** event?

Modifying HTML Elements by Using jQuery

You can use jQuery to query HTML DOM and obtain HTML elements. You can use jQuery functions to modify the attributes associated with the HTML elements. The following are some commonly used jQuery functions, which enable you to modify HTML elements:

You can use the **val** function to get or set the value of an HTML element. The following example shows how to use the **val** function to set the value of the HelloButton element to "Hello World".

jQuery functions include:

- The **val** function:
  - Allows to get or set the value of an HTML element
- The **css** function:
  - Allows to get or set the inline CSS style associated with an HTML element
- The **addclass** function:
  - Assigns the CSS class to an HTML element

The val Function

```
$('#HelloButton').val('Hello World');
```

You can use the **css** function to get or set the inline CSS style associated with an HTML element. The following example shows how to use the **css** function to set the background color of an HTML element to blue.

The css Function

```
$('#HelloButton').css('background-color','blue);
```

You can use the **addclass** function to assign a CSS class to an HTML element. The following example shows how to use the **addclass** function to add the **input_css_class** to an HTML element.

The addclass Function

```
$('#HelloButton').addClass('input_css_class');
```

Additional Reading: For more information about jQuery functions, go to http://go.microsoft.com/fwlink/?LinkID=288983&clcid=0x412

Question: If querying HTML DOM returns multiple HTML elements, how will jQuery functions handle these elements?

Calling a Web Service by Using jQuery

jQuery includes the **ajax** function that helps:

1. Perform asynchronous calls to web services.
2. Retrieve the data returned from web services.

The following example illustrates how to use the **ajax** function.

The **ajax** function:

- Helps perform calls to web services
- Helps obtain the data returned from web services
- Includes parameters such as **type**, **url**, **data**, and **contentType**

Using the ajax Function

```
var req= $.ajax({
   type: "POST",
   dataType: "json",
   url: "Customer.asmx/GetCustomerInfo",
```

```
    data: "{'ID': '123'}",
    contentType: "application/json; charset=utf-8",
    success: function (msg) {
        alert("Data Saved: " + msg);
    },
    failure: function (msg) {
        alert(msg);
    }
});
```

The **ajax** function uses the parameters **type**, **datatype**, **url**, **data**, **contentType**, **success**, and **failure** to control how to call the web services. These parameters are described as follows:

- *type*. This parameter controls the request type that you should use while querying the web services.
- *datatype*. This parameter defines the data type to be sent for AJAX services.
- *url*. This parameter provides the URL of the web services.
- *data*. This parameter defines the data that you should provide as a parameter to the web services.
- *contentType*. This parameter defines the HTTP content type that you should use, when you submit HTTP requests to web services.
- *success*. This parameter defines the name of the function, which will be triggered when the call completes successfully.
- *failure*. This parameter defines the name of the function, which will be triggered when the call completes with errors.

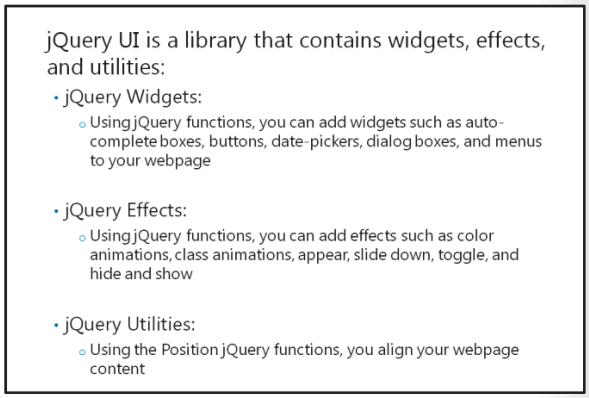
When calls to the web services complete, jQuery triggers one of two callback functions based on the success of the call.

Additional Reading: For more information about the AJAX function, go to http://go.microsoft.com/fwlink/?LinkID=288984&clcid=0x413

Question: Why should you call web services by using jQuery?

Introduction to jQueryUI

jQuery simplifies interacting with JavaScript elements, by providing a simple query-based syntax. jQuery UI is a library that includes widgets, animations, and themes that help you to build a rich user interface.



jQuery Widgets

You can add different types of widgets to your pages by using the following jQuery functions:

- *Accordion*. This function adds accordion containers to your webpage.
- *Autocomplete*. This function adds auto-complete boxes that are based on user input.
- *Button*. This function adds buttons to your webpage.

- *Datepicker*. This function adds date-pickers to your webpage.
- *Dialog*. This function adds dialog boxes to your webpage.
- *Menu*. This function adds a menu to your webpage.
- *Progressbar*. This function adds animated and static progress bars to your webpage.
- *Slider*. This function adds sliders to your webpage.
- *Spinner*. This function adds a number spinner to a data entry box.
- *Tabs*. This function adds tabs to your webpage.
- *Tooltip*. This function displays a tooltip for interface elements.

jQuery Effects

You can add various effects by using the following jQuery functions:

- *Color Animation*. This function animates colors.
- *Toggle Class*, *Add Class*, *Remove Class*, and *Switch Class*. This function adds or removes CSS classes.
- *Effect*. This function adds a variety of effects, such as appear, slide-down, explode, and fade-in.
- *Toggle*. This function toggles any jQuery effect on and off.
- *Hide and Show*. This function displays or hides any jQuery effect.

jQuery Utility

You can align your webpage content by using the **Position** jQuery utility. This utility helps align an element in relation to the position of another element.

To use jQuery UI:

1. Download the script and supporting files from http://go.microsoft.com/fwlink/?LinkID=288985&clcid=0x414
2. Reference the JavaScript libraries in the HTML files.

The following line of code shows how to use jQuery UI.

Linking to jQuery UI

```
<script src="http://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/jquery-ui.min.js"
 type="text/javascript"></script>
```

Question: What is the key difference between jQuery and jQuery UI?

Demonstration: How to Add a jQueryUI Widget

In this demonstration, you will see how to:

- Create a set of expandable sections on a webpage by using the Accordion widget.

Demonstration Steps

1. On the **DEBUG** menu of the **OperasWebsite – Microsoft Visual Studio** window, click **Start Debugging**.
2. On the Operas I Have Seen page, click the **All Operas** link.

3. In the Main Opera List section, click the **Details** link corresponding to **Cosi Fan Tutte**.
4. Under Reviews, note that there are three opera reviews displayed for **Cosi Fan Tutte**, simultaneously.
5. In the Windows Internet Explorer window, click the **Close** button.
6. In the Solution Explorer pane of the **OperasWebsite – Microsoft Visual Studio** window, under Shared, click **_ReviewsForOpera.cshtml**.
7. In the _ReviewsForOpera.cshtml code window, locate the following code.

```
<h3>Reviews</h3>
```

8. Place the mouse cursor immediately before the located code, type the following code, and then press Enter.

```
<script>
$(function() {
        $("#reviews-tool").accordion();
    });
</script>
```

9. On the **DEBUG** menu of the **OperasWebsite – Microsoft Visual Studio** window, click **Start Debugging**.
10. On the Operas I Have Seen page, click the **All Operas** link.
11. In the Main Opera List section, click the **Details** link corresponding to **Cosi Fan Tutte**.
12. Under **Reviews**, note that there are three expandable sections, and each section contains a review.

Note: You can expand each section and then read the review content.

13. In the Windows Internet Explorer window, click the **Close** button.
14. In the **OperasWebSite – Microsoft Visual Studio** window, click the **Close** button.

Lab: Using JavaScript and jQuery for Responsive MVC 4 Web Applications

Scenario

You have been asked to add a slideshow page to the web application that will show all the photos in the database. Unlike the **All Photos** gallery, which shows thumbnail images, the slideshow will display each photo in a large size. However, the slideshow will display only one photo at a time, and cycle through all the photos in the order of ID.

You want to use jQuery to create this slideshow because you want to cycle through the photos in the browser, without reloading the page each time. You also want to animate slide transitions and show a progress bar that illustrates the position of the current photo in the complete list. You will use jQueryUI to generate the progress bar.

Begin by importing a partially complete view that will display all photos simultaneously in the correct format. Then, change styles and add jQuery code to the application to create your slideshow.

Objectives

After completing this lab, you will be able to:

- Render and execute JavaScript code in the browser.
- Use the jQuery script library to update and animate page components.
- Use jQueryUI widgets in an MVC application.

Lab Setup

Estimated Time: 40 minutes

Virtual Machine: **20486B-SEA-DEV11**

User name: **Admin**

Password: **Pa$$w0rd**

Note: In Hyper-V Manager, start the **MSL-TMG1** virtual machine if it is not already running.

Before starting the lab, you need to enable the **Allow NuGet to download missing packages during build** option, by performing the following steps:

a. On the **TOOLS** menu of the Microsoft Visual Studio window, click **Options**.

b. In the navigation pane of the **Options** dialog box, click **Package Manager**.

c. Under the Package Restore section, select the **Allow NuGet to download missing packages during build** checkbox, and then click **OK**.

Exercise 1: Creating and Animating the Slideshow View

Scenario

Your team has created a view that displays photos of the right size and format. However, the view displays all photos simultaneously, one below the other.

In this exercise, you will:

- Import the view and modify the style sheet so that the photos are displayed on top of each other.

- Using jQuery, set the order for each photo so that each photo is displayed sequentially.

The main tasks for this exercise are as follows:

1. Import and test the slideshow view.
2. Modify the style sheet.
3. Animate the photo cards in the slideshow.
4. Link to the script and test the animation.

▶ Task 1: Import and test the slideshow view.

1. Start the virtual machine, and log on with the following credentials:
   - Virtual machine: **20486B-SEA-DEV11**
   - User name: **Admin**
   - Password: **Pa$$w0rd**
2. Open the **PhotoSharingApplication.sln** file from the following location:
   - File location: Allfiles (D):\Labfiles\Mod10\Starter \PhotoSharingApplication
3. Add the **SlideShow.cshtml** view file to the **Photo** folder, from the following location:
   - File location: **Allfiles (D):\Labfiles\Mod10\Slide Show View**
4. In the **PhotoController.cs** file, edit the **SlideShow** action method. Instead of throwing an exception, return the **SlideShow** view you just added. Pass a list of all the photos in the **context** object as the model.
5. Add a new site map node to the **Mvc.sitemap** file to link to the **SlideShow** action by using the following information:
   - Tag: **<mvcSiteMapNode>**
   - Title: **Slideshow**
   - Visibility: *****
   - Controller: **Photo**
   - Action: **SlideShow**
6. Start the web application in debugging mode, clear the browser cache, and then browse to the **Slideshow** view to examine the results.
7. Stop debugging.

▶ Task 2: Modify the style sheet.

1. In the **Content** folder, open the **PhotoSharingStyles.css** style sheet. Add the following properties to the style that selects ** ** tags with the **slide-show-card** class:
   - position: absolute
   - top: 0
   - **left: 0**
   - **z-index: 8**
2. Add a new style to the **PhotoSharingStyles.css** style sheet by using the following information:
   - Selector: **#slide-show DIV.active-card**

- z-index: **10**

3. Add a new style to the **PhotoSharingStyles.css** style sheet by using the following information:
   - Selector: **#slide-show DIV.last-active-card**
   - z-index: **9**
4. Start debugging, and then clear the Internet Explorer browser cache to ensure that the style sheet is reloaded.
5. Navigate to the **Slideshow** view and examine the results.
6. Stop debugging.

▶ Task 3: Animate the photo cards in the slideshow.

1. Add a new top-level folder, named **Scripts**, to the Photo Sharing application.
2. Add a new JavaScript file, **SlideShow.js**, to the **Scripts** folder.
3. In the **SlideShow.js** file, create the following global variables:
   - percentIncrement
   - percentCurrent

   Set the percentCurrent value to 100.
4. Create a new **function** named **slideSwitch** with no parameters.
5. Within the **slideSwitch** function, add a line of code that selects the first ** ** element with **active-card** class that is a child of the element with an ID of **slide-show**. Store this element in a new variable named **$activeCard**.
6. Add an **if** statement stating that if the **$activeCard** contains no elements, use the last **DIV** element with **slide-show-card** class that is a child of the element with an ID of **slide-show**.
7. Add a line of code that selects the next element after **$activeCard**. Store this element in a new variable named **$nextCard**.
8. Add an **if** statement stating that if **$nextCard** contains no elements, use the first **DIV** element with **slide-show-card** class and ID **slide-show**.
9. Add the **last-active-card** class to the **$activeCard** element.
10. Set the opacity of the **$nextCard** element to **0.0** by using the **css()** jQuery function.
11. Add the **active-card** class to the **$nextCard** element. This applies the **z-order** value **10**, from the style sheet.
12. Use the **animate()** function to fade the **$nextCard** element to opacity **1.0** over a time period of 1 second. When the **animate()** function is complete, remove the following classes from the **$activeCard** element:
    - active-card
    - last-active-card
13. Create a new anonymous function that runs when the document is fully loaded.
14. In the new anonymous function, use the **setInterval()** function to run the **slideSwitch()** function every 5 seconds.
15. Save all the changes.

▶ Task 4: Link to the script and test the animation.

1. Open the **SlideShow.cshtml** view file.
2. Add a **SCRIPT** element that links to the **SlideShow.js** script file.
3. Start the application in debugging mode and navigate to the **Slideshow** view. Observe the fade effects.
4. Stop debugging.

Results: At the end of this exercise, you will have created a Photo Sharing application with a slideshow page that displays all the photos in the application, sequentially.

Exercise 2: Optional—Adding a jQueryUI ProgressBar Widget

Scenario

The slideshow pages you added work well. Now, you have been asked to add some indication of progress through the slideshow. You want to use a progress bar to show the position of the current photo in the list of photos in the application. In this exercise, you will:

- Create a display by using the JQueryUI progress bar.
- Test the script that you created.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Complete the slideshow view and template view.
2. Modify the slideshow script.
3. Test the slideshow view.

▶ Task 1: Complete the slideshow view and template view.

1. Open the **SlideShow.cshtml** view file from the **Photo** folder.
2. Within the ** **element, add a new ** ** element with the ID **slide-show-progress-bar.**
3. Add a **<script>**tag to the **Views/Shared/_MainLayout.cshtml** view to link the view to jQuery UI. Ensure the **<script>** tag appears after the other **<script>**tags in the **HEAD** element. Link the view to the following location: **http://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/jquery-ui.min.js**

Note: In the code, note that the version number provided is 1.10.0. When typing the code, replace the version number 1.10.0 with the latest version number.

4. Add a **<link>** tag to link to the jQuery UI style sheet by using the following information:
   - Type: **text/css**
   - Rel: **stylesheet**
   - Href: **http://code.jquery.com/ui/1.9.2/themes/base/jquery-ui.css**

Note: In the code, note that the version number provided is 1.9.2. When typing the code, replace the version number 1.9.2 with the latest version number.

5. Save all the changes.

▶ Task 2: Modify the slideshow script.

1. Open the **SlideShow.js** JavaScript file.
2. Create a new function named **calculateIncrement** that takes no parameters.
3. In the new function, create a new variable named **cardCount**. Use this variable to store the number of ** ** elements within the ** ** element.
4. Divide 100 by the **cardCount** variable, and store the result in **percentIncrement**.
5. Run the jQueryUI **progressbar()** function on the ** **element. Set the **value** to 100.
6. Before the call to **setInterval()**, insert a call to the new **calculateIncrement()** function.
7. At the beginning of the **slideSwitch()** function, add the value of **percentIncrement** to the value of **percentCurrent**.
8. Add an **if** statement stating that if **percentCurrent** is more than 100, set **percentCurrent** is to equal **percentIncrement**.
9. Run the jQueryUI **progressbar()** function on the ** ** element. Set the value to **percentCurrent**.
10. Save all the changes.

▶ Task 3: Test the slideshow view.

1. Start the web application in debugging mode and clear the browser cache. Navigate to the **Slideshow** view and examine the results.
2. Stop debugging and close Visual Studio**.**

Results: At the end of this exercise, you will have created a slideshow page with a progress bar that displays the position of the current photo in the list of photos.

Question: What is the use of adding the two links to the **_MainLayout.cshtml** file in Task 1 of Exercise 2?

Question: You added **<script>** tags to the **_MainTemplate.cshtml** file to enable jQueryUI. Is this the optimal location for this link?

Module Review and Takeaways

You have seen how to use JavaScript in a web application. JavaScript helps the application interact with the actions of users and provide response to users, without reloading an entire webpage. You also saw how to use the jQuery library to access the HTML DOM structure and modify HTML elements. jQueryUI is a complement to the jQuery library, which contains functions to build rich user interfaces.

Review Question(s)

Question: You are building an application that needs to update various parts of the page every 10 seconds. Your team is proposing to use IFRAME. But you want to reduce the number of pages to be created. What type of technology should you propose to achieve this?