ASP.NET MVC 4

Exercises

Module Review and Takeaways

Table of Contents

Table of Contents i
General All Excercises v
NuGet Related Compile Errors v
Module 1 - Lab: Exploring ASP.NET MVC 4 1-1
Scenario 1-1
Objectives 1-1
Exercise 1: Exploring a Photo Sharing Application 1-1
Exercise 2: Exploring a Web Pages Application 1-2
Exercise 3: Exploring a Web Forms Application 1-3
Exercise 4: Exploring an MVC Application 1-4
Module Review and Takeaways 1-7
Module 2 - Lab: Designing ASP.NET MVC 4 Web Applications 2-1
Scenario 2-1
Objectives 2-1
Exercise 1: Planning Model Classes 2-1
Exercise 2: Planning Controllers 2-2
Exercise 3: Planning Views 2-3
Exercise 4: Architecting an MVC Web Application 2-3
Module Review and Takeaways 2-5
Module 3 - Lab: Developing ASP.NET MVC 4 Models 3-1
Scenario 3-1
Objectives 3-1
Exercise 1: Creating an MVC Project and Adding a Model 3-1
Exercise 2: Adding Properties to MVC Models 3-2
Exercise 3: Using Data Annotations in MVC Models 3-4
Exercise 4: Creating a New Windows Azure SQL Database 3-4
Exercise 5: Testing the Model and Database 3-6
Module Review and Takeaways 3-8
Module 4 - Lab: Developing ASP.NET MVC 4 Controllers 4-1
Scenario 4-1
Objectives 4-1
Exercise 1: Adding an MVC Controller and Writing the Actions 4-1
Exercise 2: Optional—Writing the Action Filters in a Controller 4-4

Exercise 3: Using the Photo Controller 4-5
Module Review and Takeaways 4-7
Module 5 - Lab: Developing ASP.NET MVC 4 Views 5-1
Scenario 5-1
Objectives 5-1
Exercise 1: Adding a View for Photo Display 5-1
Exercise 2: Adding a View for New Photos 5-2
Exercise 3: Creating and Using a Partial View 5-4
Exercise 4: Adding a Home View and Testing the Views 5-6
Module Review and Takeaways 5-8
Module 6 - Lab: Testing and Debugging ASP.NET MVC 4 Web Applications 6-1
Scenario 6-1
Objectives 6-1
Exercise 1: Performing Unit Tests 6-1
Exercise 2: Optional—Configuring Exception Handling 6-5
Module Review and Takeaways 6-8
Module 7 - Lab: Structuring ASP.NET MVC 4 Web Applications 7-1
Scenario 7-1
Objectives 7-1
Exercise 1: Using the Routing Engine 7-1
Exercise 2: Optional—Building Navigation Controls 7-4
Module Review and Takeaways 7-6
Module 8 - Lab: Applying Styles to MVC 4 Web Applications 8-1
Scenario 8-1
Objectives 8-1
Exercise 1: Creating and Applying Layouts 8-1
Exercise 2: Applying Styles to an MVC Web Application 8-4
Exercise 3: Optional—Adapting Webpages for Mobile Browsers 8-5
Module Review and Takeaways 8-7
Module 9 - Lab: Building Responsive Pages in ASP.NET MVC 4 Web Applications 9-1
Scenario 9-1
Objectives 9-1
Exercise 1: Using Partial Page Updates 9-1
Exercise 2: Optional—Configuring the ASP.NET Caches 9-4

Module Review and Takeaways 9-7
Module 10 - Using JavaScript and jQuery for Responsive MVC 4 Web Applications 10-1
Scenario 10-1
Objectives 10-1
Exercise 1: Creating and Animating the Slideshow View 10-1
Exercise 2: Optional—Adding a jQueryUI ProgressBar Widget 10-3
Module Review and Takeaways 10-5
Module 11 - Lab: Controlling Access to ASP.NET MVC 4 Web Applications 11-1
Scenario 11-1
Objectives 11-1
Exercise 1: Configuring Authentication and Membership Providers 11-1
Task 1: Configure a new Windows Azure SQL database. 11-1
Exercise 2: Building the Logon and Register Views 11-2
Exercise 3: Authorizing Access to Resources 11-6
Exercise 4: Optional—Building a Password Reset View 11-8
Module Review and Takeaways 11-11
Module 12 - Lab: Building a Resilient ASP.NET MVC 4 Web Application 12-1
Scenario 12-1
Objectives 12-1
Exercise 1: Creating Favorites Controller Actions 12-1
Exercise 2: Implementing Favorites in Views 12-2
Module Review and Takeaways 12-5
Module 13 - Lab: Using Windows Azure Web Services in ASP.NET MVC 4 Web Applications 13-1
Scenario 13-1
Objectives 13-1
Exercise 1: Accessing Windows Azure and Bing Maps 13-1
Exercise 2: Creating a WCF Service for Windows Azure 13-2
Exercise 3: Calling a Web Service from Controller Action 13-4
Module Review and Takeaways 13-7
Module 14 - Lab: Implementing APIs in ASP.NET MVC 4 Web Applications 14-1
Scenario 14-1
Objectives 14-1
Exercise 1: Adding a Web API to the Photo Sharing Application 14-1
Exercise 2: Using the Web API for a Bing Maps Display 14-3

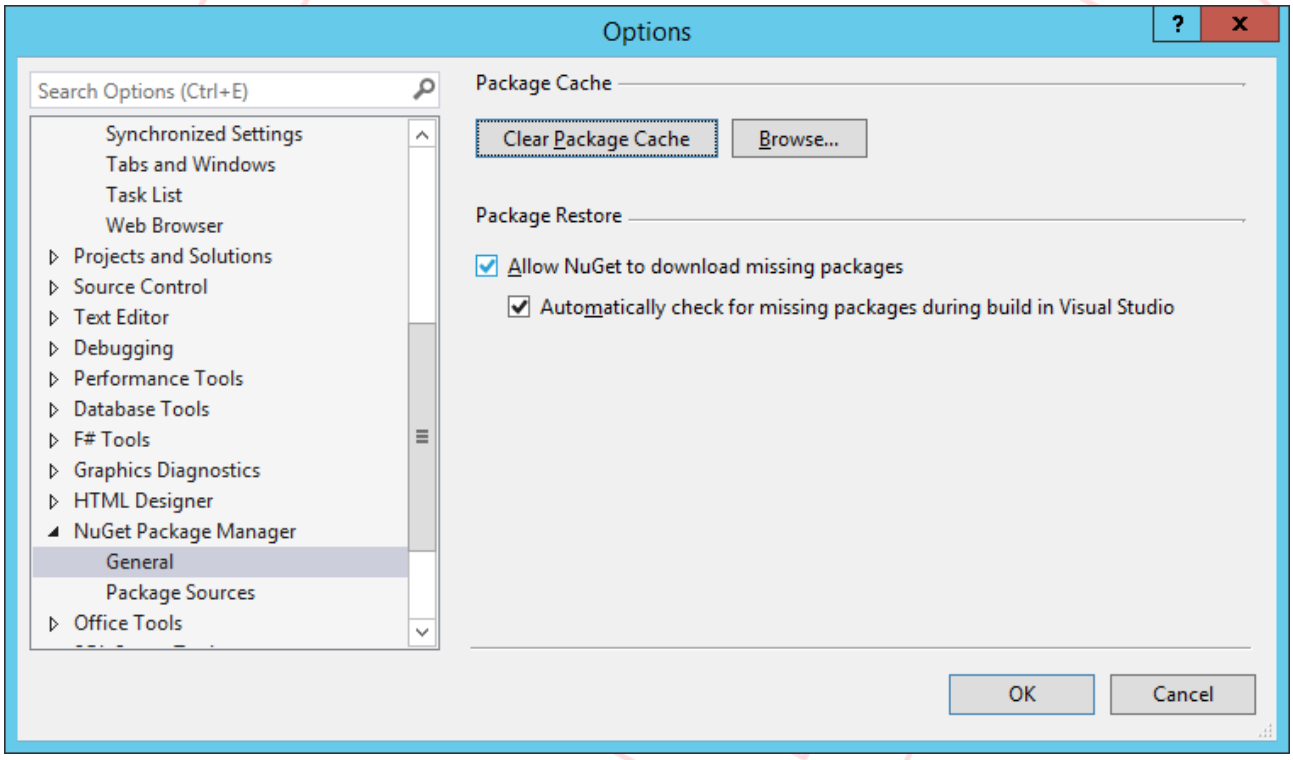
Module Review and Takeaways 14-7
Module 15 - Lab: Handling Requests in ASP.NET MVC 4 Web Applications 15-1
Scenario 15-1
Objectives 15-1
Exercise 1: Creating a SignalR Hub 15-1
Exercise 2: Creating a Photo Chat View 15-3
Module Review and Takeaways 15-6
Module 16 - Lab: Deploying ASP.NET MVC 4 Web Applications 16-1
Scenario 16-1
Objectives 16-1
Exercise 1: Deploying a Web Application to Windows Azure 16-1
Exercise 2: Testing the Completed Application 16-2
Module Review and Takeaways 16-4

General All Excercises

NuGet Related Compile Errors

If you get compile errors relating to NuGet Package manager, preform following steps:

- On the TOOLS menu of the Microsoft Visual Studio window, click Options.
- In the navigation pane of the Options dialog box, click NuGet Package Manager.
- Under the Package Restore section
    - If the 'Allow NuGet to download missing packages' and 'Automatically check for missing packages during build in Visual Studio' is selected, deselect these, and the click OK.
        - On TOOLS menu, click options, and In the navigation pane of the Options dialog box, click NuGet Package Manager, and select the 'Allow NuGet to download missing packages' and 'Automatically check for missing packages during build in Visual Studio' again, and then click OK.
    - If the 'Allow NuGet to download missing packages' and 'Automatically check for missing packages during build in Visual Studio' is not selected, the select these, and click OK:



Module 1 - 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

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. Navigate to the following location to open the PhotoSharingSample.sln file:
    - Lab\Mod01\PhotoSharingSample

2. Run the web application in non-debugging mode.
3. 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: Lab\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 2013 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 2013 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 2013 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 2 - 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

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. Open the InitialInvestigation document by using the following information:
    - File location: Lab\Mod02
2. Enable the Navigation Pane feature.
3. Read the contents of the Introduction section.
4. Read the contents of the General Description section.
5. Read the Use Cases section and then examine the Use Case Summary diagram.
6. 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: Lab\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: Lab\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: Lab\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 2013: You can create class diagrams such as LDMs in Visual Studio 2013.

Module 3 - 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 2013.
- 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

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 2013.
- 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 Visual Studio 2013 and create a new ASP.NET MVC 4 web application by using the following information:
    - Name: PhotoSharingApplication
    - Location: Lab\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 Console to add Entity Framework 5.0 to the application.
    - In the TOOL menu goto NuGet Package Manager and click the Package Manager Console
    - In the Package Manager Console type:
      Install-Package EntityFramework –Version 5.0.0
    - Press Enter.
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: Lab\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 changesmto 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:~~
~~2. Portal address: http://www.windowsazure.com/~~
~~3. User name: <your Windows Live user name>~~

4. Password: <your Windows Live password>
5. Create a new database server and a new database by using the following information:
6. Database name: PhotoSharingDB
7. Database server: New SQL Database Server
8. Login name: <your first name>
9. Login password: Pa$$w0rd
10. Login password confirmation: Pa$$w0rd
11. Region: <a region close to you>
12. In the list of allowed IP addresses for the PhotoSharingDB database, add the following IP address ranges:
13. Rule name: First Address Range
14. Start IP Address: <first address in range>
15. End IP Address: <last address in range>
16. Obtain the connection string for the PhotoSharingDB database and add it to the Web.config file.

Use following connection string:

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

17. 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:

4. Add a controller and views to the MVC web application.
5. 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:
6. Name: PhotoController
7. Template: MVC Controller with read/write actions and views, using Entity Framework
8. Model class: Photo
9. Data context class: PhotoSharingContext
10. 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: Lab\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 4 - 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:

11. Add an MVC controller to a web application.
12. Write actions in an MVC controller that respond to user operations such as create, index, display, and delete.
13. Write action filters that run code for multiple actions.

Lab Setup

Estimated Time: 60 minutes

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:

14. Index. This action gets a list of all the Photo objects and passes the list to the Index view for display.
15. Display. This action takes an ID to find a single Photo object. It passes the Photo to the Display view for display.
16. 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.
17. Create (POST). This action receives a Photo object from the Create view and saves the details to the database.
18. Delete (GET). This action displays a Photo object and requests confirmation from the user to delete the Photo object.
19. DeleteConfirmed (POST). This action deletes a Photo object after confirmation.
20. 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. Open the PhotoSharingApplication.sln file from the following location:
2. File path: Lab\Starter\PhotoSharingApplication
3. Create a new controller for handling photo objects by using the following information:
    - Controller name: PhotoController
    - Template: Empty MVC controller
4. Add using statements to the controller for the following namespaces:
    - System.Collections.Generic
    - System.Globalization
    - PhotoSharingApplication.Models
5. 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:

21. Create a temporary index and display views by using the scaffold code that is built into the Visual Studio MVC application template.
22. 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 5 - Lab: Developing ASP.NET MVC 4 Views

Scenario

You have been asked to add the following views to the photo sharing application:

o 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.

o 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.

o 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:

23. Add Razor views to an MVC application and set properties such as scaffold and model binding.
24. Write both HTML markup and C# code in a view by using Razor syntax.
25. Create a partial view and use it to display re-usable markup.

Exercise 1: Adding a View for Photo Display

Scenario

In this exercise, you will:

26. Create a new view in the Photo Sharing web application to display single photos in large size.
27. 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. Open the Photo Sharing Application solution from the following location:
    - File location: Allfiles (D):\Labfiles\Mod05\Starter\PhotoSharingApplication
2. Open the PhotoController.cs code window.
3. Build the solution
4. 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
    a. 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 reused 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 6 - 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.

Estimated Time: 90 minutes

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. Open the PhotoSharingApplication solution from the following location:
    - File location: Lab\Starter\PhotoSharingApplication
2. Add a new project to the solution for unit tests by using the following information:
    - Template: Unit Test Project
    - Name: PhotoSharingTests
3. Add a reference to the PhotoSharingApplication project in the PhotoSharingTests project.
4. 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.

1. Start debugging and display Sample Photo 5.
2. In the Internet Explorer window, request the relative URL and view the error details.
   - URL: /Photo/Display/malformedID
3. In the Internet Explorer window, request the relative URL.
   - URL: /Photo/SlideShow
4. Use the IntelliTrace pane to investigate the exception.
5. 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 welltested 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 7 - 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.

Estimated Time: 40 minutes

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. Open the PhotoSharingApplication solution from the following location:
    - File location: Lab\Starter\PhotoSharingApplication
2. 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
3. Add a reference from the Photo Sharing Tests project to the System.Web assembly.
4. Add a new Unit Test item to the PhotoSharingTests project. Name the file, RoutingTests.cs.
5. Add using statements to the RoutingTests.cs file for the following namespaces:
    - System.Web.Routing
    - System.Web.Mvc
    - PhotoSharingTests.Doubles
    - PhotoSharingApplication
6. Rename the TestMethod1 test to Test_Default_Route_ControllerOnly.
7. In the Test_Default_Route_ControllerOnly test, create a new var by using the following information:
    - Name: context
    - Type: FakeHttpContextForRouting
    - Request URL: ~/ControllerName
8. Create a new RouteCollection object named routes and pass it to the RouteConfig.RegisterRoutes() method.
9. 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
10. 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”
11. Add a new test to the RoutingTests class named, Test_Photo_Route_With_PhotoID.
12. 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
13. Create a new RouteCollection object named routes and pass it to the RouteConfig.RegisterRoutes() method.
14. 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
15. 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”

16. Add a new test to the RoutingTests class named Test_Photo_Title_Route
17. 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
18. Create a new RouteCollection object named routes and pass it to the RouteConfig.RegisterRoutes() method.
19. 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
20. 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"
21. 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 8 - 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.

Estimated Time: 40 minutes

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. Open the PhotoSharingApplication solution from the following location:

- File location: Lab\Starter\PhotoSharingApplication

2. Start the web application in debugging mode and verify that the menu and the breadcrumb trail are available on the home page.
3. Browse to the All Photos webpage and verify that the menu and the breadcrumb trail are not available on this page.
4. Browse to the Sample Photo 1 webpage and verify that the menu and the breadcrumb trail are not available on this page.
5. 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.

1. Start the web application in debugging mode.
2. Resize the browser window to the following dimensions:
    - Width: 480 pixels
    - Height: 700 pixels
3. Set the user agent string to IE9 for Windows Phone 7.
4. Refresh the home page and examine the mobile view of the application.
5. 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 9 - 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.

Estimated Time: 60 minutes

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. Open the PhotoSharingApplication.sln file from the following location:
    - File location: Lab\Starter\PhotoSharingApplication
2. Create a new folder in the Views folder by using the following information:
    - Name of the new folder: Comment
3. Add an existing item to the new Comment folder by using the following information:
    - File location of the existing item: Lab\Comment Components\Delete.cshtml
4. Add an existing item to the Controller folder by using the following information:
    - File location of the existing item: Lab\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
    - Parameter: an integer named PhotoId

- 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.
- Save the PhotoId parameter value in the ViewBag collection to use it later in the view.
- Return a partial view as the result of the _CommentsForPhoto action by using the following information:
    - View name: _CommentsForPhoto
    - Model: comments.ToList()
- 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.
- Bind the _CommentsForPhoto.cshtml view to an enumerable collection of comments.
- Create an H3 element by using the following information:
- Heading: Comments
- 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.
- For each item in the model, render a DIV element with the photo-comment class.
- Within the element, add a DIV element with the photo-commentfrom class. Within this DIV element, render the UserName value of the model item by using the Html.DisplayFor() helper.
- 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.
- 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.
- Render a link to the Delete action by using the Html.ActionLink() helper. Pass the item.CommentID value as the id parameter.
- 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
- PhotoId parameter: Model.PhotoID
- Run the application in debugging mode and browse to Sample Photo 1. Observe the display of comments on the page.
- 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()

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 thefo llowing 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

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

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. Open the PhotoSharingApplication.sln file from the following location:
    - File location: Lab\Starter\PhotoSharingApplication
2. Add the SlideShow.cshtml view file to the Photo folder, from the following location:
    - File location: Lab \Slide Show View

3. 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.
4. 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
5. Start the web application in debugging mode, clear the browser cache, and then browse to the Slideshow view to examine the results.
6. 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 activecard 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?

Module 11 - Lab: Controlling Access to ASP.NET MVC 4 Web Applications

Scenario

A large part of the functionality for your proposed Photo Sharing application is in place. However, stakeholders are concerned about security because there are no restrictions on the tasks that users can complete. The following restrictions are required:

o Only site members should be able to add or delete photos.

o Only site members should be able to add or delete comments.

You have been asked to resolve these concerns by creating a membership system for the Photo Sharing application. Visitors should be able to register as users of the web application and create user accounts for themselves. After registration, when the users log on to the application, they will have access to actions such as adding and deleting photos and comments. Anonymous users will not have access to perform these actions. Additionally, registered users should also be able to reset their own password.

Objectives

After completing this lab, you will be able to:

- Configure a web application to use ASP.NET Form Authentication with accounts stored in Local database.
- Write models, controllers, and views to authenticate users in a web application.
- Provide access to resources in a web application.
- Enable users to reset their own password.

Estimated Time: 90 minutes

Exercise 1: Configuring Authentication and Membership Providers

Scenario

You want to use a local database to store user accounts and membership information.

In this exercise, you will:

- Configure a provider to connect to the database.

The main tasks for this exercise are as follows:

1. Install universal providers.
2. Configure providers in Web.config.

Task 1: Configure a new Windows Azure SQL database.

Task 1: Install universal providers.

1. Open the PhotoSharingApplication.sln file from the following location:
    - File location: Lab \Starter\PhotoSharingApplication
2. Or Continue with your exsisting PhotoSharingApplication
3. Install the Microsoft ASP.NET Universal Providers package in the PhotoSharingApplication project by using the NuGet Package Manager.

Task 3: Configure providers in Web.config.

1. Remove the connection string named DefaultConnection from the top-level Web.config file.
2. Obtain the connection string for the PhotoAppServices database and add it to the Web.config file.
3. Configure the web application to use Forms authentication in Web.config, by using the following information:
    - Parent element: <system.web>
    - Element: <authentication>
    - Mode: Forms
4. Configure the logon page for the web application by using the following information:
    - Parent element: <authentication>
    - Element: <forms>
    - Logon URL: ~/Account/Login
    - Timeout: 2880
5. Configure the default profile provider to use the connection string named PhotoAppServices.
6. Configure the Default Membership Provider to use the connection string named PhotoAppServices.
7. Configure the Default Role Provider to use the connection string named PhotoAppServices.
8. Configure the Default Session Provider to use the connection string named PhotoAppServices.
9. Save all the changes.

Results: At the end of this exercise, you will have created a Photo Sharing application that is configured to use Windows Azure SQL database for user accounts and membership information. In subsequent exercises, you will add model classes, actions, and views to implement authentication for this database.

Exercise 2: Building the Logon and Register Views

Scenario

You have configured the Photo Sharing application to connect to Windows Azure SQL database for authentication and membership services. However, to use forms authentication in an MVC application, you need to build model classes, controllers, and views that enable users to log on, log off, and register for an account.

In this exercise, you will:

- Add model classes.
- Add controllers.
- Import logon and register views.
- Test the developed components.

The main tasks for this exercise are as follows:

1. Add account model classes.
2. Add an account controller.
3. Import Logon and Register views.
4. Add authentication controls to the Template view.
5. Test registration, log on, and log off.

Task 1: Add account model classes.

1. Add a new Class file named AccountModelClasses.cs to the Models folder.
2. Add references to the following namespaces, to the new class file:
    - System.ComponentModel.DataAnnotations

- System.Data.Entity

3. Remove the AccountModelClasses class and add a new class by using the following information:
   - Scope: Public
   - Name: UsersContext
   - Inherit: DbContext
4. In the UsersContext class, create a constructor that passes the PhotoAppServices connection string to the base class constructor.
5. In the AccountModelClasses.cs code file, add a new public class named Login.
6. Add a new property to the Login class by using the following information:
   - Scope: public
   - Type: string
   - Name: UserName
   - Access: Read/Write
   - Display name: User name
   - Use the Required annotation.
7. Add a new property to the Login class by using the following information:
   - Scope: public
   - Type: string
   - Name: Password
   - Access: Read/Write
   - Data type: Password
   - Use the Required annotation.
8. Add a new property to the Login class by using the following information:
   - Scope: public
   - Type: bool
   - Name: RememberMe
   - Access: Read/Write
   - Display name: Remember me?
9. In the AccountModelClasses.cs code file, add a new public class named Register.
10. Add a new property to the Register class by using the following information:
    - Scope: public
    - Type: string
    - Name: UserName
    - Access: Read/Write
    - Display name: User name
    - Use the Required annotation.
11. Add a new property to the Register class by using the following information:
    - Scope: public
    - Type: string
    - Name: Password
    - Access: Read/Write
    - Data type: Password
    - Use the Required annotation.
12. Add a new property to the Register class by using the following information:
    - Scope: public
    - Type: string

- Name: ConfirmPassword
- Access: Read/Write
- Data type: Password
- Display name: Confirm password
- Ensure that this property matches the Password property by using the Compare annotation.

13. Save all the changes.

Task 2: Add an account controller.

1. Add a new controller named AccountController to the MVC web application by using the Empty MVC controller template.
2. Delete the default Index action from the AccountController file and add using statement references for the following namespaces:
   - System.Web.Security
   - PhotoSharingApplication.Models
3. Create a new action method in the AccountController class by using the following information:
   - Scope: public
   - Return type: ActionResult
   - Name: Login
   - Parameter: a string named returnUrl
4. In the Login action, store the returnUrl value in the ViewBag collection, and then return a view named Login.
5. Create a new action method in the AccountController class by using the following information:
   - HTTP verb: POST
   - Scope: public
   - Return type: ActionResult
   - Name: Login
   - Parameters: a Login object named model and a string named returnUrl
6. Within the Login action method code block for the HTTP POST verb, check if the ModelState is valid.
7. Add an if…else statement to check the user's credentials by using the following information:
   - Method: Membership.ValidateUser
   - User name: model.UserName
   - Password: model.Password
8. If the user's credentials are correct, authenticate the user by using the following information:
   - Method: FormsAuthentication.SetAuthCookie
   - User name: model.UserName
   - Create persistent cookie: model.RememberMe
9. If returnUrl is a local URL, redirect the user to the returnUrl. Otherwise, redirect the user to the Index action of the Home controller.
10. If the user's credentials are incorrect, add a model error to the ModelState object by using the following information:
    - Key: An empty string
    - Error message: The username or password is incorrect
11. If the ModelState is not valid, return the current view and pass the model object so that the user can correct errors.
12. Create a new action method in the AccountController class by using the following information:
    - Scope: public

- Return type: ActionResult
- Name: LogOff
- Parameters: None

13. In the LogOff action, log off the user, and then redirect to the Index action of the Home controller by using the FormsAuthentication.SignOut() method.
14. Create a new action method in the AccountController class by using the following information:
    - Scope: public
    - Return type: ActionResult
    - Name: Register
    - Parameters: None
15. In the Register action, return the Register view.
16. Create a new action method in the AccountController class by using the following information:
    - HTTP verb: POST
    - Scope: public
    - Return type: ActionResult
    - Name: Register
    - Parameter: a Register object named model.
17. Within the Register action method code block for the HTTP POST verb, check if the ModelState is valid.
18. If the ModelState is valid, create a try…catch block that catches exceptions of the type MembershipCreateUserException e.
19. In the try block, create a new user with the right user name and password by using the Membership.CreateUser method. Store the result in a MembershipUser object named NewUser.
20. Authenticate the new user and redirect the browser to the Index action of the Home controller.
21. In the catch block, add a model error to the ModelState object by using the following information:
    - Key: Registration Error
    - Error message: Report the error status code as a string
22. If the ModelState is not valid, return the current view and pass the model object so that the user can correct errors.
23. Save all the changes.

Task 3: Import Logon and Register views.

1. Add a new folder named Account to the Views folder.
2. Add the Login.cshtml file to the Account folder from the following location:
    - File location: Allfiles (D):\Labfiles\Mod11\Account Views
3. Add the Register.cshtml file to the Account folder from the following location:
    - File location: Allfiles (D):\Labfiles\Mod11\Account Views

Task 4: Add authentication controls to the Template view.

1. Open the _MainLayout.cshtml page for editing.
2. Immediately before the DIV element with clear-floats class, insert a DIV element with logincontrols class.
3. In the new DIV element, write a Razor if… else code block that checks whether the current request is from an authenticated user.
4. If the request is from an authenticated user, render a greeting message that includes the authenticated user's name.
5. After the greeting message, render a link to the LogOff action by using the following information:
    - Helper: Html.ActionLink()
    - Link text: Log Off

- Action name: LogOff
- Controller name: Account

6. If the request is from an anonymous user, render a link to the Login action by using the following Information:
   - Helper: Html.ActionLink()
   - Link text: Log In
   - Action name: Login
   - Controller name: Account
7. After the Log In link, render a link to the Register action by using the following Information:
   - Helper: Html.ActionLink()
   - Link text: Register
   - Action name: Register
   - Controller name: Account
8. Save all the changes.

Task 5: Test registration, log on, and log off.

1. Start the web application in debugging mode and register a user account by using the following information:
   - User name: David Johnson
   - Password: Pa$$w0rd
2. Log off and then log on with the credentials you just created.
3. Stop debugging.

Results: At the end of this exercise, you will create a Photo Sharing application in which users can register for an account, log on, and log off.

Exercise 3: Authorizing Access to Resources

Scenario

Now that you have enabled and tested authentication, you can authorize access to resources for both anonymous and authenticated users.

You should ensure that:

- Only site members can add or delete photos.
- Only site members can add or delete comments.
- The account controller actions are authorized properly.
- Only authenticated users see the_Create view for comments in the Display view.

The main tasks for this exercise are as follows:

1. Restrict access to Photo actions.
2. Restrict access to the Comment actions.
3. Restrict access to the Account actions.
4. Check authentication status in a view.
5. Test authorization.

Task 1: Restrict access to Photo actions.

1. In the PhotoController.cs file, add the [Authorize] annotation to ensure that only authenticated users can access the Create action for the GET requests.
2. Add the [Authorize] annotation to ensure that only authenticated users can access the Create action for the HTTP POST verb.
3. Add the [Authorize] annotation to ensure that only authenticated users can access the Delete action.
4. Add the [Authorize] annotation to ensure that only authenticated users can access the DeleteConfirmed action for the HTTP POST verb.
5. Save all the changes.

Task 2: Restrict access to the Comment actions.

1. In the CommentController.cs file, add the [Authorize] annotation to ensure that only authenticated users can access the _Create action.
2. Add the [Authorize] annotation to ensure that only authenticated users can access the Delete action.
3. Add the [Authorize] annotation to ensure that only authenticated users can access the DeleteConfirmed action for the HTTP POST verb.
4. Save all the changes.

Task 3: Restrict access to the Account actions.

1. In the AccountController.cs file, add the [Authorize] annotation to ensure that only authenticated users can access all actions by default.
2. Add the [AllowAnonymous] annotation to ensure that anonymous users can access the Login action.
3. Add the [AllowAnonymous] annotation to ensure that anonymous users can access the Login action for the HTTP POST verb.
4. Add the [AllowAnonymous] annotation to ensure that anonymous users can access the Register action.
5. Add the [AllowAnonymous] annotation to ensure that anonymous users can access the Register action for the HTTP POST verb.
6. Save all the changes.

Task 4: Check authentication status in a view.

1. Open the _CommentsForPhoto.cshtml partial view.
2. In the _CommentsForPhoto.cshtml partial view, add an if statement to ensure that the _Create partial view is only displayed if the request is authenticated.
3. If the request is not authenticated, render a link to the Login action of the Account controller to display the text To comment you must log in.
4. Save all the changes.

Task 5: Test authorization.

1. Start the web application in debugging mode and then attempt to add a new photo to the web application, without logging on to the application.
2. Without logging on to the application, view any photo in the application and attempt to add a comment.
3. Log on to the web application by using the following credentials:
    - User name: David Johnson
    - Password: Pa$$w0rd
4. Add a comment of your choice to the photo by using the following information:
    - Subject: Authenticated Test Comment
5. Stop debugging.

Results: At the end of this exercise, you will have authorized anonymous and authenticated users to access resources in your web application.

Exercise 4: Optional—Building a Password Reset View

Scenario

Site visitors can now register as users of the Photo Sharing application and log on to the site so that they can add photos and comments. However, they do not have the facility to change their password. In this exercise, you will create a password reset page by using the membership services provider.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Add a local password model class.
2. Add reset password actions.
3. Import the reset password view.
4. Add a link to the reset password view.
5. Test password reset.

Task 1: Add a local password model class.

1. Add a new class to the AccountModelClasses.cs file by using the following information:
    - Scope: public
    - Name of the class: LocalPassword
2. Add a new property to the LocalPassword class by using the following information:
    - Scope: public
    - Type: string
    - Name: OldPassword
    - Access: Read/Write
    - Data type: Password
    - Annotation: [Required]
    - Display name: Current password
3. Add a new property to the LocalPassword class by using the following information:
    - Scope: public
    - Type: string
    - Name: NewPassword
    - Access: Read/Write
    - Data type: Password
    - Annotation: [Required]
    - Display name: New password
4. Add a new property to the LocalPassword class by using the following information:
    - Scope: public
    - Type: string
    - Name: ConfirmPassword
    - Access: Read/Write
    - Data type: Password
    - Display name: Confirm new password
    - Use the Compare annotation to ensure this property matches the NewPassword property.
5. Save all the changes.

Task 2: Add reset password actions.

1. In the AccountController class, add a new enumeration by using the following information:
    - Scope: public
    - Name: ManageMessageId
    - Values: ChangePasswordSuccess, SetPasswordSuccess
2. Add a new action to the AccountController class by using the following information:
    - Scope: public
    - Return type: ActionResult
    - Name: ResetPassword
    - Parameters: an optional ManageMessageId object named message
3. If the message parameter is not null, set the ViewBag.StatusMessage property to Your password has been changed.
4. Set the ViewBag.ReturnUrl property to the URL of the ResetPassword action.
5. Return a view named ResetPassword.
6. Add a new action to the AccountController class by using the following information:
    - Annotation: HttpPost
    - Scope: public
    - Return type: ActionResult
    - Name: ResetPassword
    - Parameter: a LocalPassword object named model
7. In the new ResetPassword action for the HTTP POST verb, set the ViewBag.ReturnUrl property to the URL of the ResetPassword action.
8. Include an if statement to check whether the ModelState is valid.
9. If the ModelState is valid, create a new Boolean variable named changePasswordSucceeded, and then add a try… catch block that catches all exceptions.
10. In the try block, change the user's password by using the following information:
    - Method: Membership.Provider.ChangePassword
    - User name: User.Identity.Name
    - Old password: model.OldPassword
    - New password: model.NewPassword
    - Store the result in changePasswordSucceeded
11. In the catch block, set the changePasswordSucceeded variable to false.
12. After the try…catch code block, if changePasswordSucceeded is true, redirect to the ResetPassword action and pass the ManageMessageId.ChangePasswordSucccess value to the message parameter.
13. If changePasswordSucceeded is false, add a model error to the ModelState object by using the following information:
    - Key: An empty string
    - Message: The current password is incorrect or the new password is invalid
14. If the ModelState is not valid, return the current view and pass the model object so that the errors can be corrected.
15. Save all the changes.

Task 3: Import the reset password view.

1. Add the ResetPassword.cshtml file to the Views/Account folder from the following location:
    - Lab\Account Views

Task 4: Add a link to the reset password view.

1. Add a new link to the _MainLayout.cshmtl template view by using the following information:
    - Position: After the link to the LogOff action
    - Text: Reset
    - Action: ResetPassword
    - Controller: Account
2. Save all the changes.

Task 5: Test password reset.

1. Start the web application in debugging mode, and then log on with the following credentials:
    - User name: David Johnson
    - Password: Pa$$w0rd
2. Change the password from Pa$$w0rd to Pa$$w0rd2.
3. Stop debugging and close the open windows.

Results:

At the end of this exercise, you will build a Photo Sharing application in which registered users can reset their own password.

Question:

In Exercise 3, when you tried to add a photo before logging on to the application, why did ASP.NET display the Login view?

Question:

How can you ensure that only Adventure Works employees are granted access to the Delete action of the Photo controller?

Module Review and Takeaways

In this module, you discussed the various membership and role providers in ASP.NET 4.5. You compared the benefits of using SimpleProviders and UniversalProviders with the benefits of using other providers, for database-centric applications. However, for applications that need to use Windows for authentication and authorization, you can use the existing role and membership providers. You also viewed how to implement custom providers, to add functionalities such as password encryption to your web application.

Review Question(s)

Question: What is the key difference between implementing authentication and authorization in ASP.NET 4.5 from implementing the same in the previous versions of ASP.NET?

Real-world Issues and Scenarios

When you create web applications, you may need to create custom providers because you do not want to use the schema provided by Microsoft. However, you can use SimpleProviders to remove the need to develop custom providers and reduce the effort required for building applications.

Module 12 - Lab: Building a Resilient ASP.NET MVC 4 Web Application

Scenario

The senior developer has asked you to implement the following functionality in your Photo Sharing web application.

- Any visitor of the application, including anonymous users, should be able to mark a photograph as a favorite.
- If a user has marked a favorite, a link should be available to display the favorite photo.
- Favorite photos should be displayed in the slideshow view.

Objectives

After completing this lab, you will be able to:

- Store a setting for an anonymous or authenticated user in session state.
- Check a user preference when rendering an action link.
- Render a webpage by checking state values in the application.

Lab Setup

Estimated Time: 45 minutes

Before starting the lab, you need to perform the following step:

- Download Manage Nuget packages from the Project menu and note the version number for the jquery.ui package.

Exercise 1: Creating Favorites Controller Actions

Scenario

You have been asked to build functionality that stores the favorite photos of the visitors in the session state of the web application. After users add photos to their favorites, they will be able to view a slideshow of all the photos they selected as favorites.

In this exercise, you will:

- Create the Favorites Slideshow action.
- Create the Add Favorite action.

The main tasks for this exercise are as follows:

1. Create the Favorites Slideshow action.
2. Create the Add Favorite action.

Task 1: Create the Favorites Slideshow action.

1. Open the PhotoSharingApplication.sln file from the following location:
    - File location: Lab\Starter\PhotoSharingApplication
2. In the PhotoController.cs code file, add a new action by using the following information:
    - Scope: public
    - Return type: ActionResult
    - Name: FavoritesSlideShow

3. In the FavoritesSlideShow action, create and instantiate a new enumerable list of Photo objects named favPhotos.
4. Create a new enumerable list of integers named favoriteIds. Set this list to be equal to the Session["Favorites"] variable by casting this variable as a list of integers.
5. If favoriteIds is null, set favoriteIds to be a new enumerable list of integers.
6. Create a new Photo object named currentPhoto. Do not instantiate the new object.
7. Create a new foreach code block that loops through all the integers in the favoriteIds list.
8. For each integer in the favoriteIds list, obtain the Photo object with the right ID value by using the context.FindPhotoById() method. Store the object in the currentPhoto variable.
9. If the currentPhoto variable is not equal to null, then add currentPhoto to the favPhotos list.
10. At the end of the FavoritesSlideShow action, return the SlideShow view and pass the favPhotos list as a model class.
11. Save all the changes.

Task 2: Create the Add Favorite action.

1. Add a new action to the PhotoController.cs file by using the following information:
    - Scope: public
    - Return type: ContentResult
    - Name: AddFavorite
    - Parameter: an integer named PhotoId
2. Create a new enumerable list of integers named favoriteIds. Set this list to be equal to the Session["Favorites"] variable, by casting the variable as a list of integers.
3. If favoriteIds is null, set favoriteIds to be a new enumerable list of integers.
4. Add the PhotoId value to the favoriteIds list of integers.
5. Set the Session["Favorites"] variable to equal the favoriteIds variable.
6. Return HTML content by using the following information:
    - Method: Content()
    - Content: The picture has been added to your favorites
    - Content type: text/plain
    - Encoding: System.Text.Encoding.Default
7. Save all the changes.

Results: After completing this exercise, you will be able to create controller actions that store values in the session state of the web application, and retrieve values from the same session state.

Exercise 2: Implementing Favorites in Views

Scenario

You have created the necessary controller actions to implement favorite photos. Now, you should implement the user interface components to display a control for adding a favorite. If a user has favorites, you should display a link to the FavoritesSlideShow action.

In this exercise, you will:

- Add an AJAX action link in the Photo Display view.
- Add a link and update the site map.

The main tasks for this exercise are as follows:

1. Add an AJAX action link in the Photo Display view.
2. Add a link and update the site map.
3. Test favorites.

Task 1: Add an AJAX action link in the Photo Display view.

1. Open the Display.cshtml view for the Photo controller.
2. At the end of the element, insert a new element with the ID, addtofavorites.
3. In the DIV element, include the Ajax.ActionLink() helper to render a link to the AddFavorite action by using the following information:
    - Text: Add this photo to your favorites
    - Action: AddFavorite
    - Controller: Photo
    - Route values: Pass the Model.PhotoID value to the PhotoId action parameter.
    - Pass a new AjaxOptions object.
    - Update target ID: addtofavorites
    - HTTP method: GET
    - Insertion mode: Replace

4. Save all the changes.

Task 2: Add a link and update the site map.

1. Open the _MainLayout.cshtml view for editing.
2. After the breadcrumb trail, add a Razor @if statement to ensure that the Session["Favorites"] variable is not null.
3. If the Session["Favorites"] variable is not null, render a element with the ID, favorites-link.
4. In the DIV element, render a link to the FavoritesSlideShow action by using the following information:
    - Helper: Html.ActionLink()
    - Link text: Favorite Photos
    - Action: FavoritesSlideShow
    - Controller: Photo
5. Save all the changes.
6. Open the Mvc.sitemap file.
7. Add a new site map node to the Mvc.sitemap file by using the following information:
    - Title: Favorites
    - Visibility: SiteMapPathHelper,!*
    - Controller: Photo
    - Action: FavoritesSlideShow
8. Save all the changes.

Task 3: Test favorites.

1. Start the web application in debugging mode.

2. Add three photos of your choice to your favorite photos.

3. Browse to the home page and click the Favorite Photos link.

4. Stop debugging and close Visual Studio.

Results: After completing this exercise, you will be able to :
Create the user interface components for the favorite photos functionality.
Test the functionality of the user interface components.

Question:

In this lab, you stored the list of favorite photos in the session state. While testing, your manager notices that authenticated users lose their favorite photos list whenever they close their browser. Where would you store a list of favorites for each authenticated user so that the list is preserved whenever a user logs on to the web application?

Question:

How would you create a view of favorite photos with the card-style presentation users see on the All Photos page?

Module Review and Takeaways

Web applications are usually subject to different kinds of attacks. These attacks enable attackers to access sensitive information stored in the database and to perform malicious actions. You can use AntiXSS and request validation, to protect the applications from web attacks. You can also use state management techniques to store information for multiple HTTP requests to help avoid users from typing the same information more than once.

Review Question(s)

Question:

Recently, an error occurred in one of applications that you had developed for your company. After performing few tests, you realize that the issue was due to an HTML code that was passed from the user to the server. You want to prevent such issues in the future. What would you consider to do?

Real-world Issues and Scenarios

While implementing web applications, you may want to use a rich format input editor, to enable users to format the input within text boxes. Therefore, you may need to disable request validation, to enable ASP.NET to capture and process user input.

Complex business functions usually involve multiple views. Such functions can pose problems because information must be shared across multiple views. Session state management helps resolve these problems, because it enables retaining information pertinent to multiple views

Module 13 - Lab: Using Windows Azure Web Services in ASP.NET MVC 4 Web Applications

Scenario

In the Photo Sharing application, the users have the option to add location information of a photo when they upload it. The senior developer recommends that you should store the location as a longitude and latitude, and an address so that other applications can use the data in mash-ups. You have been asked to create a service, hosted in Windows Azure, which will perform this conversion. You have to call this service from the Photo Upload page in the Photo Sharing application.

Objectives

After completing this lab, you will be able to:

- Install the Windows Azure software development kit (SDK).
- Create a Bing Maps developer account and trial key.
- Write a web service in Visual Studio that is hosted in Windows Azure.
- Call a Windows Azure web service from server-side code in a web application.

Lab Setup

Estimated Time: 75 minutes

Exercise 1: Accessing Windows Azure and Bing Maps

Scenario

To develop a Windows Communication Foundation (WCF) service that is hosted in Windows Azure, you must install the Windows Azure SDK. To resolve address details to latitude and longitude data, you will use the Bing Maps Location API. You will call this API from the WCF service hosted in Windows Azure so that you can re-use your web service from other Adventure Works websites and applications.

In this exercise, you will:

- Install the Windows Azure SDK.
- Create a Bing Maps developer account.
- Create a Bing Maps Key.

The main tasks for this exercise are as follows:

1. Install the Windows Azure SDK.
2. Create a Bing Maps developer account.
3. Create a Bing Maps key.

Task 1: Install the Windows Azure SDK.

1. Navigate to the following webpage:
   - http://www.microsoft.com/web/downloads/platform.aspx
2. Download and run the Web Platform Installer.
3. Use the Web Platform Installer to download and install the Windows Azure SDK for .NET (VS 2013).

Task 2: Create a Bing Maps developer account.

1. Navigate to the following webpage:
   - https://www.bingmapsportal.com
2. Log on to the Bing Maps Account Center web application by using following the Windows Live account credentials:
   - User name: <Your Windows Live account name>
   - Password: <Your Windows Live account password>
3. Register a new Bing Maps developer account by using the following information:
   - Account Name: <Your account name>
   - Contact Name: <Your name>
   - Email Address: <Your Windows Live account name>

Task 3: Create a Bing Maps key.

1. Create a new Bing Maps key in the Photo Sharing application by using the following information:
   - Application name: Photo Sharing Application
   - Key type: Trial
   - Application type: Public website

Results: After completing this exercise, you will be able to register an application to access the Bing Maps APIs.

Exercise 2: Creating a WCF Service for Windows Azure

Scenario

You want to create a WCF service to resolve a locality string to a latitude and longitude by looking up the information in the Bing Maps Geocode SOAP service. After creating the WCF service, you need to host the service in Windows Azure by using the Windows Azure Cloud Service project template from the Windows Azure SDK.

In this exercise, you will:

- Add a new Windows Azure Cloud Service project to the web application.
- Create the Location Checker Service interface.
- Add a service reference to the Bing Maps Geocode service.
- Write the Location Checker service.
- Publish the Service in Windows Azure.

The main tasks for this exercise are as follows:

1. Add a new Windows Azure Cloud Service project to the web application.
2. Create the Location Checker Service interface.
3. Add a service reference to the Bing Maps Geocode service.
4. Write the Location Checker service.
5. Publish the Service in Windows Azure.

Task 1: Add a new Windows Azure Cloud Service project to the web application.

1. Open the PhotoSharingApplication.sln file from the following location:
   - File location: Lab\Starter\PhotoSharingApplication
2. Add a new project to the PhotoSharingApplication web application by using the following information:

- Template: Windows Azure Cloud Service
- Name: LocationChecker
- .NET Framework 4.5
- Click Ok
- role: WCF Service Web Role

3. Rename the WCFServiceWebRole1 project as LocationCheckerWebRole.
4. Rename the IService1.cs file as ILocationCheckerService.cs.
5. Rename the Service1.svc file as LocationCheckerService.svc.
6. Rename the WCFServiceWebRole1 namespace as LocationCheckerWebRole throughout the LocationCheckerService.svc.cs file.
7. Rename the Service1 class as LocationCheckerService.
8. In the ServiceDefinition.csdef file, set the value of vmsize for the LocationCheckerWebRole to ExtraSmall.
9. Save all the changes.

Task 2: Create the Location Checker Service interface.

1. Remove all the method declarations from the ILocationCheckerService interface.
2. Add a new method declaration to the ILocationCheckerService interface by using the following information:
   - Annotation: [OperationContract]
   - Return type: string
   - Name: GetLocation
   - Parameter: a string named address
3. Save all the changes.

Task 3: Add a service reference to the Bing Maps Geocode service.

1. Add a service reference to the LocationCheckerWebRole project by using the following information:
   - Address: http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc?wsdl.
   - Namespace: GeocodeService
2. In the LocationCheckerService.svc code file, add a using statement for the following namespace:
   - LocationCheckerWebRole.GeocodeService
3. Save all the changes.

Task 4: Write the Location Checker service.

1. Remove the following public methods from the LocationCheckerService class:
   - GetData
   - GetDataUsingDataContract
2. Add a new method to the LocationCheckerService class by using the following information:
   - Scope: public
   - Return type: string
   - Name: GetLocation
   - Parameter: a string named address
3. In the GetLocation method, create a string variable, results, and initialize the value of the results variable to an empty string.
4. From the Bing Maps Account Center page, copy the key you created for the Photo Sharing Application.

5. In the GetLocation method, create a string variable, key, and paste the key from the Bing Maps Account Center page as a value of the key variable.
6. Open the Geocoding Code.txt from the following location:
   - Lab\Service Code
7. Copy all the text from the Geocoding Code.txt file and paste the text into the GetLocation method, immediately after the key variable initialization.
8. At the end of the GetLocation method, return the string variable results.

~~Task 5: Publish the Service in Windows Azure.~~

~~1. Package the LocationChecker project by using the following information:~~

- ~~Service configuration: Cloud~~
- ~~Build configuration: Debug~~
- ~~In Windows Internet Explorer, log on to the Windows Azure Portal.~~

~~2. Create a new cloud service by using the following information:~~

- ~~URL: <Your Windows Live Account Name>LocationService~~
- ~~Region: <Choose a location near you>~~

~~Note: If your Windows Live Account Name includes dots or @ symbols, replace these characters with dashes.~~

~~3. Upload a new staging deployment to the new cloud service by using the following information:~~

- ~~Name: LocationChecker~~
- ~~Package location: Lab\Starter\PhotoSharingApplication\LocationChecker\bin\Debug\app.publish~~
- ~~Package: LocationChecker.cspkg~~
- ~~Configuration: ServiceConfiguration.Cloud.cscfg~~
- ~~Deploy even if one or more roles contain a single instance.~~

~~Results: After completing this exercise, you will be able to create a WCF cloud service. You will also be able to publish the WCF cloud service in Windows Azure.~~

Exercise 3: Calling a Web Service from Controller Action

Scenario

Now that you have created and deployed the Location Checker WCF service in Windows Azure, you can call the service from the Photo Sharing ASP.NET MVC web application. You can also call the service from other .NET code, such as desktop applications, if necessary.

In this exercise, you will use the Location Checker service to add latitude and longitude data to new photos as they are added to the Photo Sharing application.

The main tasks for this exercise are as follows:

1. Add a service reference to the Photo Sharing application.
2. Call the WCF service from the photo create action.
3. Configure the Services Database connection string.
4. Test the Location Checker service.

Task 1: Add a service reference to the Photo Sharing application.

1. Add a new Service Reference to the PhotoSharingApplication project by using the following information:
   - Click Discover

- Select LocaltionCheckerService.svc
- Namespace: GeocodeService
- Click OK

2. Add a using statement for the following namespace to the PhotoController.cs code file:
   - PhotoSharingApplication.GeocodeService
3. Save all the changes.

Task 2: Call the WCF service from the photo create action.

1. Add a new method to the PhotoController class by using the following information:
   - Scope: private
   - Return type: string
   - Name: CheckLocation
   - Parameter: a string named location
2. In the CheckLocation method, create a new variable by using the following information:
   - Type: LocationCheckerServiceClient
   - Name of the variable: client
   - Value of the variable: Null
3. In the CheckLocation method, create a new string variable, response, and initialize the value of the response variable to Null.
4. Add a new try…catch statement that catches all errors in a variable named e.
5. In the try block, set the client variable to be a new LocationCheckerServiceClient.
6. Pass the location parameter to the client.GetLocation method and store the result in the response variable.
7. In the catch block, store the e.Message property in the response variable.
8. At the end of the CheckLocation method, return the response string.
9. At the start of the Create action method for the HTTP POST verb, add an if statement that checks that photo.Location is not an empty string.
10. In the if statement, pass the photo.Location property to the CheckLocation method and store the result in a new string variable named stringLongLat.
11. Add an if statement to check whether the stringLongLat variable starts with the string, Success.
12. In the if statement, create a new array of characters named splitChars and add the single character ":" to the array.
13. Pass the splitChars array to the Split method of stringLongLat, and store the result in a new array of strings named coordinates.
14. Set the photo.Latitude property to the second string in the coordinates array and set the photo.Longitude property to the third string in the coordinates array.
15. Save all the changes.

Task 4: Test the Location Checker service.

1. Set PhotoSharingApplication as the startup project for the solution.
2. Run the web application in debugging mode, and then log on to the web application by using the following credentials:
   - User name: David Johnson
   - Password: Pa$$w0rd2
3. Add a new photo to the application by using the following information:
   - Title: Testing Locations

- Photo: Lab\Sample Photos\Beach.jpg
- Location: Florence, OR

4. Browse to the Home page and display your new photo to check the Latitude and Longitude properties.
5. Stop debugging and close Visual Studio.

Results

After completing this exercise, you will be able to add a service reference for a Windows Azure WCF service to a .NET Framework application. You will also be able to call a WCF service from an MVC controller action.

Question

What is the advantage of calling the Bing Maps Geocoding service from a WCF service in Windows Azure, instead of calling the Geocoding service directly from the Create action in the MVC web application?

Question

Why is latitude and longitude data useful for photos in the Photo Sharing application?

Module Review and Takeaways

You can upload applications and services on Windows Azure to enable better scalability and re-usability of the application logic. Windows Azure includes different roles that support application needs such as hosting and storage. You can also use cloud services to host WCF services, and use the hosting services available in MVC and HTML applications.

Review Question(s)

Question

Your teammate enquired whether you would be using Windows Azure or self host the server for the application. What should you recommend to your teammate based on the fact that the application loading would be seasonal?

Module 14 - Lab: Implementing APIs in ASP.NET MVC 4 Web Applications

Scenario

Your manager wants to ensure that the photos and information stored in the Photo Sharing application can be integrated with other data in web mash-ups, mobile applications, and other locations. To re-use such data, while maintaining security, you need to implement a RESTful Web API for the application. You will use this Web API to display the locations of photos on a Bing Maps page.

Objectives

After completing this lab, you will be able to:

- Create a Web API by using the new features of ASP.NET MVC 4.
- Add routes and controllers to an application to handle REST requests.
- Call a REST Web API from jQuery client-side code.

Lab Setup

Estimated Time: 60 minutes

Exercise 1: Adding a Web API to the Photo Sharing Application

Scenario

You have been asked to implement a Web API for the Photo Sharing application to ensure that photos can be used in third-party websites, mobile device applications, and other applications. In this exercise, you will

- Add a Web API controller for the Photo model class.
- Configure formatters and routes to support the Web API.
- Test the API by using Internet Explorer.

The main tasks for this exercise are as follows:

1. Add a Photo API controller.
2. Configure API routes.
3. Configure media-type formatters.
4. Test the Web API with Internet Explorer.

Task 1: Add a Photo API controller.

1. Open the PhotoSharingApplication solution from the following location:
   - File location: Labfiles\Starter\PhotoSharingApplication
2. Add a new API controller to the PhotoSharingApplication project by using the following information:
   - Name: PhotoApiController
   - Template: Empty API controller
3. Add a using statement for the following namespace to PhotoApiController.cs:
   - PhotoSharingApplication.Models
4. Add a new variable to the PhotoApiController class by using the following information:
   - Scope: private
   - Type: IPhotoSharingContext
   - Name: context

- Initial value: a new PhotoSharingContext object

5. Add a new action to the PhotoApiController by using the following information:
   - Scope: public
   - Return type: IEnumerable<Photo>
   - Name: GetAllPhotos
   - Parameters: none
6. In the GetAllPhotos action, return the context.Photos collection as an enumerable object.
7. Add a new action to the PhotoApiController by using the following information:
   - Scope: public
   - Return type: Photo
   - Name: GetPhotoById
   - Parameters: an integer named id
8. In the GetPhotoById action, pass the id parameter to the context.FindPhotoById() method. Store the returned Photo object in a variable named photo.
9. If the photo variable is null, throw a new HttpResponseException and pass the HttpStatusCode.NotFound value.
10. At the end of the GetPhotoById action, return the photo object.
11. Add a new action to the PhotoApiController by using the following information:
    - Scope: public
    - Return type: Photo
    - Name: GetPhotoByTitle
    - Parameters: a string named title
12. In the GetPhotoByTitle action, pass the title parameter to the context.FindPhotoByTitle() method. Store the returned Photo object in a variable named photo.
13. If the photo variable is null, throw a new HttpResponseException and pass the HttpStatusCode.NotFound value.
14. At the end of the GetPhotoByTitle action, return the photo object.
15. Save all the changes.

Task 2: Configure API routes.

1. In the WebApiConfig.cs code file, in the Register method, remove all existing route registrations.
2. Add a new route to the Register method by using the following information:
   - Name: PhotoApi
   - Route template: api/photos/{id}
   - Default controller: PhotoApi
   - Default action: GetPhotoById
   - Constraint: id= "[0-9]+"
3. After the PhotoApi route, add a new route to the Register method by using the following information:
   - Name: PhotoTitleApi
   - Route template: api/photos/{title}
   - Default controller: PhotoApi
   - Default action: GetPhotoByTitle
4. After the PhotoTitleApi route, add a new route to the Register method by using the following information:
   - Name: PhotosApi

- Route template: api/photos
- Default controller: PhotoApi
- Default action: GetAllPhotos

5. Save all the changes.

Task 3: Configure media-type formatters.

1. In the WebApiConfig.cs code file, at the end of the Register method, create a new variable named json. Set this variable to config.Formatters.JsonFormatter.
2. Set the json.SerializerSettings.PreserveReferencesHandling property to Newtonsoft.Json.PreserveReferencesHandling.Objects.
3. Remove the XmlFormatter object from the config.Formatters collection.
4. Save all the changes.

Task 4: Test the Web API with Internet Explorer.

1. Start that web application in debugging mode.
2. Request the photo with ID 4 by using the Web API. Display the returned JSON file by using Visual Studio and check that the Title property is Sample Photo 4.
3. Request the photo with title Sample Photo 5 by using the Web API. Display the returned JSON file by using Visual Studio and check that the Title property is Sample Photo 5.
4. Request all photos by using the Web API. Display the returned JSON file by using Visual Studio, and check that both Sample Photo 9 and Sample Photo 13 are present.
5. Close Visual Studio and stop debugging.

Results: At the end of this exercise, you will be able to create a simple Web API for an ASP.NET MVC 4 web application.

Exercise 2: Using the Web API for a Bing Maps Display

Scenario

You need to use the new Web API to obtain the photos in the client-side jQuery code. You will use latitude and longitude properties to display these photos as pins on a Bing API map.

To create the map display in the Photo Sharing application, you must add a new view and action for the photo controller. You must also add a new template view because the Bing Maps AJAX control requires a different <!DOCTYPE> directive to the one in use elsewhere in the Photo Sharing application. You will import a JavaScript file with basic Bing Maps code in place. To this JavaScript file, you will add code to call the Web API, obtain photo details, and display them on the map.

In this exercise, you will:

- Create a new template view.
- Create a map action, view, and script file.
- Obtain and display photos.
- Test the Bing Maps control.

The main tasks for this exercise are as follows:

1. Create a new template view.
2. Create a map action, view, and script file.

3. Obtain and display photos.
4. Test the Bing Maps control.

Task 1: Create a new template view.

1. In the Views/Shared folder, create a copy of the _MainLayout.cshtml view file and name the copy as _MapLayout.cshtml.
2. In the _MapLayout.cshtml file, replace the <!DOCTYPE html> declaration with <!DOCTYPE html PUBLIC "-//W3C/DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">.
3. In the onload event of the BODY element, call the GetMap() JavaScript function.
4. Remove the links to jQueryUI and unobtrusive AJAX.
5. Add a new link to the Bing Maps AJAX control in the HEAD element by using the following information:

- Charset: UTF-8
- Type: text/javascript
- SRC: http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=7.0

6. Save all the changes.

Task 2: Create a map action, view, and script file.

1. Add a new action to the PhotoController class by using the following information:

- Scope: public
- Return type: ViewResult
- Name: Map
- Parameters: none

2. In the new Map action, return a View named Map.
3. Add a new view for the Map action by using the following information:

- View name: Map
- Type: Do not create a strongly-typed view
- Layout: _MapLayout.cshtml

4. Remove the H2 element from the Map.cshmtl view.
5. Add a new JavaScript block to the Map.cshtml view.
6. In the new JavaScript block, create a new variable named webApiUrl by using the Url.Content() helper to set the webApiUrl variable to ~/api/photos.
7. Create a new variable named pictureUrl by using the Url.Action() helper to set the pictureUrl variable to the URL of the GetImage action of the Photo controller. Include a forward slash at the end of the URL.
8. Create a new variable named displayUrl by using the Url.Action() helper to set the displayUrl variable to the URL of the Display action of the Photo controller. Include a forward slash at the end of the URL.
9. Add a new JavaScript script, with an empty src attribute, to the Map.cshtml view.
10. Use the Url.Content() helper to set the src attribute in the new JavaScript code to ~/Scripts/MapDisplay.js.
11. Create a new DIV element by using the following information:

- ID: mapDiv
- Style position: absolute
- Style width: 650px
- Style height: 400px

12. Add the MapDisplay.js JavaScript file to the Scripts folder from the following location:

- Allfiles (D):\Labfiles\Mod14\Bing Maps Script

13. Add a new node to the site map by using the following information:
- Title: Map
- Visibility: *
- Controller: Photo
- Action: Map
14. Save all the changes.
15. Start the web application in debugging mode and browse to the Map page.
16. Stop debugging.

Task 3: Obtain and display photos.

1. In the MapDisplay.js script file, add a new function by using the following information:
- Name: GetPhotos
- Parameter: serviceUrl
2. Set the $.support.cors value to true.
3. Use the $.ajax() jQuery function to call the GetPhotos Web API by using the following information:
- URL: serviceUrl
- Type: GET
- Data type: json
- Success: DisplayPics
- Error: OnError
4. Add a new function by using the following information:
- Name: DisplayPics
- Parameter: response
5. In the DisplayPics function, create two variables named location and pin.
6. Use the jQuery $.each function to loop through all the photo objects in the response collection.
7. For each photo object in the response collection, set the location variable to a new location by using the following information:
- Object: Microsoft.Maps.Location
- Latitude: photo.Latitude
- Longitude: photo.Longitude
8. Set the pin variable to a new push pin by using the following information:
- Object: Microsoft.Maps.Pushpin
- Location: location
9. Set the pin.Title property to photo.Title and the pin.ID property to photo.PhotoID.
10. Ensure that the DisplayInfoBox method handles the click event for pushpins by using the following information:
- Method: Microsoft.Maps.Events.addHandler
- Object: pin
- Event: 'click'
- Handler method: DisplayInfoBox
11. Add the pin object to the dataLayer object by using the dataLayer.push function.
12. Add a new function by using the following information:
- Name: OnError
- Parameter: response

13. In the OnError function, use the alert function to inform the user that the picture coordinates could not be obtained.
14. At the end of the GetMap function, call the GetPhotos function and pass the webApiUrl variable.
15. Save all the changes.

Task 4: Test the Bing Maps control.

1. Start the web application in debugging mode and browse to the Map page to check the map control.
2. Click a pin of your choice.
3. Click the thumbnail.
4. Stop debugging and close Visual Studio.

Results

After completing this exercise, you will be able to create a template view to display a Bing Map AJAX control, and create a view and script file to display a Bing Map. You will also use jQuery to call a Web API and obtain details of photos. You will then mash up the data from a web API with Bing Maps data.

Question

How do the API actions you added to the PhotoApiController controller in Exercise 1 differ from other actions in MVC controllers?

Module Review and Takeaways

You can use the Web API framework to facilitate creating REST-style Web API calls in applications. RESTstyle Web API is recommended for mobile applications because of the light weight design of REST services.

REST services use HTTP methods such as GET, POST, and PUT to notify the API of the action that it needs to perform. Web APIs use the media formatter and the JSON.NET library to serialize and deserialize information, respectively. You can call Web API services by using server-side code, jQuery code, and the JSON.NET library.

Real-world Issues and Scenarios

Consider that you develop a mobile application by using Web APIs and the application needs to use currency rate services. For this application, you cannot use WCF, because WCF can impede the performance of the application by using XML for data exchanges. Therefore, you should use REST and JSON in the application to reduce the data that is transmitted between the client system and the server.

Review Question(s)

Question: We are developing a mobile application, which requires to access business data via internet. You are proposing to use Web API but your colleague is proposing to use WCF.

What would be the key point for using Web API in this scenario?

Module 15 - Lab: Handling Requests in ASP.NET MVC 4 Web Applications

Scenario

The Adventures Works board and managers are pleased with the Photo Sharing application, but have requested that interactivity should be maximized to encourage users to register and participate fully in the community. Therefore, you have been asked to add chat functionality to the application.

Authenticated members should be able to start a chat on a particular photo from the Display view. Chat rooms for each photo should be separated from each other. Users in the chat room should be able to send a message to all other users in that chat room, and they should be able to see all the messages that have been sent since they joined the chat room.

You have decided to use SignalR to implement the chat room over Web Sockets.

Objectives

After completing this lab, you will be able to:

- Install SignalR in an ASP.NET MVC 4 web application.
- Configure SignalR on the server and create a SignalR hub.
- Link to the required script files for SignalR in an MVC view.
- Create the script for SignalR connections and send messages to groups.

Lab Setup

Estimated Time:

60 minutes

Exercise 1: Creating a SignalR Hub

Scenario

Before you can write JScript code on the client to connect to SignalR, you must configure and code a SignalR Hub on the web server.

In this exercise, you will:

- Install SignalR in the Photo Sharing application.
- Configure routing.
- Create a SignalR hub to accept messages from clients and forward those messages to other clients who are chatting about the same photo.

The main tasks for this exercise are as follows:

1. Install SignalR.
2. Create a Hub class.
3. Configure SignalR routes.

Task 1: Install SignalR.

1. Open the PhotoSharingApplication.sln file from the following location:
   - File location: Lab\Starter\PhotoSharingApplication

2. In the NuGet Package Manager, search for the following pre-release package:
   - Microsoft.aspnet.signalr
3. Install the Microsoft ASP.NET SignalR package in the PhotoSharingApplication project.
4. Notice the additions that the NuGet package has made to the project references and the Scripts folder.

Task 2: Create a Hub class.

1. Add a new class file named ChatHub.cs to the PhotoSharingApplication.
2. Remove the following namespace references from the ChatHub.cs class file:
   - System.Collections.Generic;
   - System.Linq;
3. Add the following namespace references to the ChatHub.cs class file:
   - System.Threading.Tasks
   - Microsoft.AspNet.SignalR
4. Ensure that the ChatHub class inherits from the Microsoft.AspNet.SignalR.Hub class.
5. In the ChatHub class, create a new method by using the following information:
   - Scope: public
   - Return type: Task
   - Name: Join
   - Parameter: an integer named photoId
6. In the Join method, return the result of the Groups.Add() method by using the following information:
   - Connection ID: Context.ConnectionId
   - Group name: "Photo" + photoId
7. In the ChatHub class, create a new method by using following information:
   - Scope: public
   - Return type: Task
   - Name: Send
   - First parameter: a string named username
   - Second parameter: an integer named photoId
   - Third parameter: a string named message
8. In the Send method, create a new string variable named groupName and set the value to "Photo" + photoId.
9. In the Send method, return the result of the addMessage method for the groupName group by using the following information:
   - Method: Clients.Group(groupName).addMessage()
   - User name: username
   - Message: message
10. Save all the changes.

Task 3: Configure SignalR routes.

1. In the Global.asax code-behind file, add a reference to the Microsoft.AspNet.SignalR namespace.
2. In the Application_Start() method, immediately after the RegisterAllAreas() code, call the RouteTable.Routes.MapHubs() method.
3. Save all the changes.

Results: After completing this exercise, you will be able to install SignalR in an MVC 4 web application, configure routes for SignalR, and create a hub to receive and forward simple text messages.

Exercise 2: Creating a Photo Chat View

Scenario

Now that you have set up and configured SignalR and a SignalR hub on the server side, you must use JScript and the SignalR JScript library to send and receive messages on the client side.

In this exercise, you will:

- Create a new MVC controller action and Razor view to display the chat user interface for a particular photo.
- Link to the JScript libraries that SignalR requires and write client-side script to call the Join() and Send() methods on the hub.
- Test the chat functionality.

The main tasks for this exercise are as follows:

1. Create a chat action and view.
2. Link to the chat view.
3. Link to JScript files.
4. Script SignalR connections.
5. Script SignalR messages.
6. Test the chat room.

Task 1: Create a chat action and view.

1. Add a new action to the PhotoController class by using the following information:

- Annotation: [Authorize]
- Scope: public
- Return type: ActionResult
- Name: Chat
- Parameter: an integer named id

2. Create a new Photo object named photo and get the photo value by passing the id parameter to the context.FindPhotoById() method.
3. If the photo object is null, return an HTTP Not Found status code.
4. At the end of the action, return the Chat view and pass the photo object as the model class.
5. Add the following view file to the Views/Photo folder:

- Allfiles (D):\Labfiles\Mod15\Chat View\Chat.cshtml

6. Save all the changes.

Task 2: Link to the chat view.

1. In the Display.cshtml view file, after the DIV element with ID addtofavorites, add a new DIV element, with ID chataboutthisphoto.
2. In the new DIV element, render a link to the Chat view by using the following information:

- Helper: Html.ActionLink()
- Text: Chat about this photo
- View: Chat

- Route values: pass the Model.PhotoID value to the id parameter.

3. Start the web application in the debugging mode, display a photo of your choice, and then click Chat.
4. Log on with the following credentials:

- User name: David Johnson
- Password: Pa$$w0rd2

5. Attempt to send a chat message.
6. Stop debugging.

Task 3: Link to JScript files.

1. Add a new SCRIPT element at the end of the Chat.cshtml view file, with type text/javascript.
2. In the new SCRIPT element, create a new variable named username. In a new Razor code block, use the User.Identity.Name property to set the value for the variable.
3. Create a second new variable named photoid and use the Model.PhotoID property to set the value for the variable.
4. Add a new SCRIPT element, with type text/javascript, and an empty src attribute.
5. In the new script element, use the Url.Content() helper to set the src attribute to the following path:

- ~/Scripts/jquery.signalR-1.0.0.js

Note: NuGet installs the latest version of SignalR. Ensure that the name of the script file you enter matches the name of the file in the Scripts folder.

6. Add a new SCRIPT element, with type text/javascript, and an empty src attribute.
7. In the new script element, use the Url.Content() helper to set the src attribute to the following path:

- ~/signalr/hubs

8. Add a new SCRIPT element, with type text/javascript, and an empty src attribute.
9. In the new script element, use the Url.Content() helper to set the src attribute to the following path:

- ~/Scripts/ChatRoom.js

10. Save all the changes.

Task 4: Script SignalR connections.

1. Add a new JScript file named ChatRoom.js to the Scripts folder.
2. In the new JavaScript file, use jQuery to create an anonymous function that runs when the page loads.
3. Use the $.connection.chatHub property to obtain the ChatHub you already created, and then store the hub in a variable named chat.
4. Use jQuery to set the initial focus on the element with ID chat-message.
5. Create an anonymous function that runs when the $.connection.hub.start() method is done.
6. Call the chat.server.join() function on the SignalR hub, pass the photoid value as a parameter, and then create an anonymous function that runs when the function is done.
7. Save all the changes.

Task 5: Script SignalR messages.

1. In the ChatRoom.js JScript file, after creating and instantiating the chat variable, set the chat.client.addMessage property to be a new anonymous function with two parameters, name and message.
2. In the new function, create a variable named encodedName, and use jQuery to set this variable to a new with the name parameter as its HTML content.
3. In the new function, create a variable named encodedMessage, and use jQuery to set this variable to a new with the message parameter as its HTML content.

4. Create a new variable named listItem. Set the value of this variable to an HTML LI element that includes the encodedName and encodedMessage variables.
5. Append the listItem element to the page element with ID discussion.
6. In the function that runs when the client.server.join() method is done, create an anonymous function that runs when the button with ID sendmessage is clicked.
7. In the new anonymous function, call the chat.server.send() method by using the following information:

- User name: username
- Photo ID: photoid
- Message: use the value of the element with ID chat-message

8. Use jQuery to obtain the element with ID chat-message, set its value to an empty string, and give it the focus.
9. Save all the changes.

Task 6: Test the chat room.

1. Start the web application in debugging mode and log on with the following credentials:

- User name: David Johnson
- Password: Pa$$w0rd2

2. Browse to the chat page for Sample Photo 1.
3. Send a message of your choice and observe the results.
4. Start a new instance of Windows Internet Explorer, and browse to the Photo Sharing application home page.
5. Register a new user account with the following credentials:

- User name: Mark Steele
- Password: Pa$$w0rd

6. Browse to the chat page for Sample Photo 1, and then send a message of your choice.
7. Switch to the first instance of Internet Explorer, and then send a second message of your choice. Observe the messages sent between the two users.
8. Stop debugging and close Visual Studio.

Results

After completing this exercise, you will be able to create MVC controller actions and views to display a user interface for SignalR functionality, link to the JScript libraries that SignalR requires, and use JScript to connect to a SignalR hub and send messages.

Question

In the chat functionality that you created, each photo in the Photo Sharing application has a separate chat room. How is this separation possible with one SignalR hub?

Question

In Exercise 2, you wrote JScript code that called the chat.server.join() and chat.server.send() functions. In which script file are these functions defined?

Module Review and Takeaways

Most web applications do not limit themselves to rendering HTML content. They require functionalities that support custom logic such as custom authentication. To suit such cases, you can use HTTP modules and HTTP handlers to run custom logic before or after rendering a webpage. The web socket technology in MVC 4 provides support for two-way communication between client and server systems. This technology is useful for web applications that require constant updates between the client and server systems.

Real-world Issues and Scenarios

You create an application that obtains the latest product pricing information from the internal database.

The pricing is constantly updated every time business users feel the need for updates. Therefore, you need to ensure that the application updates pricing information every five minutes. In such cases, you can to use the web socket technology to implement price update. You can also add code to download the product image stored in the product database by using a generic handler (*.ashx file).

Review Question(s)

Question: You want to select technology that could be used for developing a web based chatting applications for your client. Which technology would you prefer in this scenario?

Module 16 - Lab: Deploying ASP.NET MVC 4 Web Applications

Scenario

You have completed the development and testing of the photo sharing application. Your managers and senior developers have signed off the project, and have requested you to deploy the application to the Adventure Works Windows Azure account.

Objectives

After completing this lab, you will be able to:

- Prepare an MVC web application for deployment.
- Deploy an MVC web application to Windows Azure.

Lab Setup

Estimated Time: 45 minutes

In this lab, you will create a web application in Windows Azure that runs in Free mode. Free mode is an excellent tool for testing and running small, non-intensive web applications. However, there is a limit of 165 megabytes (MB) of data transfer per day on the Free mode tool. After completing Exercise 2, you are encouraged to further test the deployed web application. However, if you do a lot of extra testing, you may encounter the limit and the application may become unavailable.

Exercise 1: Deploying a Web Application to Windows Azure

Scenario

In this exercise, you will:

- Reconfigure the Photo Sharing application for release deployment.
- Configure the Entity Framework initializer class, which fills the database with initial data, and ensure that the build configuration and connection strings are correct.
- Create a new web application in Windows Azure and deploy the Photo Sharing application to the new site.

The main tasks for this exercise are as follows:

1. Prepare the Photo Sharing application project for deployment.
2. Create a new website in Windows Azure.
3. Deploy the Photo Sharing application.

Task 1: Prepare the Photo Sharing application project for deployment.

1. Open the Photo Sharing application solution from the following location:
   - File location: Allfiles (D):\Labfiles\Mod16\Starter\PhotoSharingApplication.
2. In the Build properties of the PhotoSharingApplication project, select the Release configuration.
3. Log on to the Windows Azure portal by using the following information:
   - URL: http://www.windowsazure.com
   - User name: <Your Windows Live account name>
   - Password: <your password>
4. Copy the ADO.NET connection string for the PhotoSharingDB to the clipboard.

5. In the Web.config file, paste the connection string into the connectionString attribute of the PhotoSharingDB connection string. Set the password in the pasted connection string to Pa$$w0rd.
6. Log on to the Bing Maps Account Center web application by using the following information:
   - URL: https://www.bingmapsportal.com
   - User name: <Your Windows Live account name>
   - Password: <Your Windows Live account password>
7. Copy the Bing Maps key.
8. Replace the text, {Your Bing Key}, in the MapDisplay.js file, with the copied key.
9. Save all the changes.

Task 2: Create a new website in Windows Azure.

1. In Windows Azure, create a new website by using the following information:
   - URL: <your username>PhotoSharing
   - Region: <a region near you>
   - Database: PhotoSharingDB
   - Database connection string name: PhotoSharingDB.
   - Database user: <Your first name>
   - Password: Pa$$w0rd

Task 3: Deploy the Photo Sharing application.

1. In Windows Internet Explorer, download the publish profile for the <your username>PhotoSharing web application.
2. In Microsoft Visual Studio, start the Publish Web wizard and import the publish profile you just downloaded.
3. For the Photo Sharing Context database connection, select the PhotoSharingDB connection string.
4. For the Photo Sharing DB database connection, select the PhotoSharingDB connection string.
5. For the UsersContext database connection, select the PhotoSharingDB connection string.
6. Preview the files that require changes, and then publish the web application.

Results: After completing this exercise, you will be able to prepare an ASP.NET MVC web application for production deployment, create a web application in Windows Azure, and deploy an MVC web application to Windows Azure.

Exercise 2: Testing the Completed Application

Scenario

You have completed and fully deployed the Photo Sharing web application in Windows Azure. Now, you want to perform some final functionality tests before you confirm the completion of the application to your manager.

The main tasks for this exercise are as follows:

1. Add a photo and a comment.
2. Use the slideshow and favorites options.
3. Test the chat room.

Task 1: Add a photo and a comment.

1. Log on to the Adventure Works Photo Sharing web application by using the following information:
   - User name: David Johnson

- Password: Pa$$w0rd2

2. Add a new photo to the web application by using the following information:
   - Title: Test New Photo
   - Photo: Allfiles (D):\Labfiles\Mod16\Sample Photos\track.JPG
   - Description: This is the first photo added to the deployed web application
3. Display the new photo and check if the data is displayed correctly.
4. Add a new comment to the new photo by using the following information:
   - Subject: Functionality Check
   - Body: The photo metadata is displayed correctly

Task 2: Use the slideshow and favorites options.

1. Examine the Slideshow webpage and check that the display of all photos functions as designed.
2. Add three photos to your favorites list.
3. View the Favorites slideshow.

Task 3: Test the chat room.

1. Enter the chat room for Sample Photo 1.
2. Send a message.
3. Start a new instance of Internet Explorer and browse to the Photo Sharing application by using the following information:
   - URL: http://<yourusername>PhotoSharing.azurewebsites.net
   - User name: Mark Steele
   - Password: Pa$$w0rd
4. Enter the chat room for Sample Photo 1.
5. Send a second message and switch to the first instance of Internet Explorer. Send a third message.
6. Close Internet Explorer and Visual Studio.

Results: After completing this exercise, you will be able to confirm that all the functionalities that you built in the Photo Sharing application run correctly in the Windows Azure deployment.

Question

Why is it unnecessary to use bin deployment in this lab?

Question

In the labs for this course, you used the same Windows Azure SQL Database for both development and production. If you wanted to use separate databases for development and production, but did not want to reconfigure the web application every time you deployed to the development and production web servers, how would you configure the web application?

Module Review and Takeaways

Deployment is usually the last task that developers perform; however, this is the task that developers do not spend much time on. The target environment usually impacts the deployment procedure. In the case of Windows Azure, service packages allow the management portal to deploy the package to the virtual machine that hosts the workload.

To simplify the deployment process, Microsoft Visual Studio consists of a number of tools that help developers generate files for deployment. These tools also help handle the database during deployment.

You can also generate the Web.config file based on the environment that it deploys to.

Review Question(s)

Question: You need to create two packages for deployment-for testing and for production environment. This is because of the difference in server name and other environment settings. What should you do?