



OFFICIAL MICROSOFT LEARNING PRODUCT

20486D

Developing ASP.NET Core MVC Web
Applications

Contents

Module 1: Exploring ASP.NET Core MVC

Module Overview	01-1
Lesson 1: Overview of Microsoft Web Technologies	01-3
Lesson 2: Overview of ASP.NET 4.x	01-12
Lesson 3: Introduction to ASP.NET Core MVC	01-22
Lab: Exploring ASP.NET Core MVC	01-29
Module Review and Takeaways	01-31

Module 2: Designing ASP.NET Core MVC Web Applications

Module Overview	02-1
Lesson 1: Planning in the Project Design Phase	02-2
Lesson 2: Designing Models, Controllers and Views	02-15
Lab: Designing ASP.NET Core MVC Web Applications	02-22
Module Review and Takeaways	02-24

Module 3: Configure Middleware and Services in ASP.NET Core

Module Overview	03-1
Lesson 1: Configuring Middleware	03-2
Lesson 2: Configuring Services	03-14
Lab: Configuring Middleware and Services in ASP.NET Core	03-28
Module Review and Takeaways	03-30

Module 4: Developing Controllers

Module Overview	04-1
Lesson 1: Writing Controllers and Actions	04-2
Lesson 2: Configuring Routes	04-10
Lesson 3: Writing Action Filters	04-26
Lab: Developing Controllers	04-32
Module Review and Takeaways	04-34

Module 5: Developing Views

Module Overview	05-1
Lesson 1: Creating Views with Razor Syntax	05-2
Lesson 2: Using HTML Helpers and Tag Helpers	05-12
Lesson 3: Reusing Code in Views	05-20
Lab: Developing Views	05-29
Module Review and Takeaways	05-31

Module 6: Developing Models

Module Overview	06-1
Lesson 1: Creating MVC Models	06-2
Lesson 2: Working with Forms	06-13
Lesson 3: Validating MVC Application	06-24
Lab: Developing Models	06-31
Module Review and Takeaways	06-33

Module 7: Using Entity Framework Core in ASP.NET Core

Module Overview	07-1
Lesson 1: Introduction to Entity Framework Core	07-2
Lesson 2: Working with Entity Framework Core	07-10
Lesson 3: Using Entity Framework Core to Connect to Microsoft SQL Server	07-22
Lab: Using Entity Framework Core in ASP.NET Core	07-35
Module Review and Takeaways	07-37

Module 8: Using Layouts, CSS and JavaScript in ASP.NET Core MVC

Module Overview	08-1
Lesson 1: Using Layouts	08-2
Lesson 2: Using CSS and JavaScript	08-8
Lesson 3: Using jQuery	08-19
Lab: Using Layouts, CSS and JavaScript in ASP.NET Core MVC	08-29
Module Review and Takeaways	08-31

Module 9: Client-Side Development

Module Overview	09-1
Lesson 1: Applying Styles	09-2
Lesson 2: Using Task Runners	09-19
Lesson 3: Responsive Design	09-33
Lab: Client-Side Development	09-42
Module Review and Takeaways	09-44

Module 10: Testing and Troubleshooting

Module Overview	10-1
Lesson 1: Testing MVC Applications	10-2
Lesson 2: Implementing an Exception Handling Strategy	10-14
Lesson 3: Logging MVC Applications	10-24
Lab: Testing and Troubleshooting	10-31
Module Review and Takeaways	10-34

Module 11: Managing Security

Module Overview	11-1
Lesson 1: Authentication in ASP.NET Core	11-2
Lesson 2: Authorization in ASP.NET Core	11-18
Lesson 3: Defending from Attacks	11-28
Lab: Managing Security	11-45
Module Review and Takeaways	11-47

Module 12: Performance and Communication

Module Overview	12-1
Lesson 1: Implementing a Caching Strategy	12-2
Lesson 2: Managing State	12-11
Lesson 3: Two-Way Communication	12-20
Lab: Performance and Communication	12-32
Module Review and Takeaways	12-34

Module 13: Implementing Web APIs

Module Overview	13-1
Lesson 1: Introducing Web APIs	13-2
Lesson 2: Developing a Web API	13-9
Lesson 3: Calling a Web API	13-21
Lab: Implementing Web APIs	13-30
Module Review and Takeaways	13-32

Module 14: Hosting and Deployment

Module Overview	14-1
Lesson 1: On-Premises Hosting and Deployment	14-2
Lesson 2: Deployment to Microsoft Azure	14-19
Lesson 3: Microsoft Azure Fundamentals	14-29
Lab: Hosting and Deployment	14-42
Module Review and Takeaways	14-44

Notes

About This Course

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

Course Description

In this course, students will learn to develop advanced Microsoft ASP.NET Core Model-View-Controller (MVC) applications. The focus will be on coding activities that enhance the performance and scalability of the website application.

Audience

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

Student Prerequisites

This course requires that you meet the following prerequisites:

- Experience with Visual Studio 2017.
- Experience with C# programming and concepts such as Lambda expressions, LINQ, and anonymous types.
- Experience in using the Microsoft .NET Framework.
- Experience with HTML, CSS, and JavaScript.
- Experience with querying and manipulating data with ADO.NET.
- Knowledge of XML and JSON data structures.

Course Objectives

After completing this course, students will be able to:

- Describe the Microsoft Web Technologies stack and select an appropriate technology to use to develop any given application.
- Design the architecture and implementation of a web application that will meet a set of functional requirements, user interface requirements, and address business models.
- Configure the pipeline of ASP.NET Core web applications by using middleware and leverage dependency injection across MVC applications.
- Add controllers to an MVC application to manage user interaction, update models, and select and return views.
- Develop a web application that uses the ASP.NET Core routing engine to present friendly URLs and a logical navigation hierarchy to users.
- Create views, which display and edit data and interact with models and controllers, in an MVC application.
- Create MVC models and write code that implements business logic within model methods, properties, and events.
- Connect an ASP.NET Core application to a database by using Entity Framework Core.
- Implement a consistent look and feel across an entire MVC web application.

- Write JavaScript code that runs on the client-side and utilizes the jQuery script library to optimize the responsiveness of an MVC web application.
- Add client side packages and configure Task Runners.
- Run unit tests and debugging tools against a web application in Visual Studio 2017.
- Write an MVC application that authenticates and authorizes users to access content securely by using Identity.
- Build an MVC application that resists malicious attacks.
- Use caching to accelerate responses to user requests.
- Use SignalR to enable two-way communication between client and server.
- Describe what a Web API is and why developers might add a Web API to an application.
- Describe how to package and deploy an ASP.NET Core MVC web application from a development computer to a web server.

Course Outline

The course outline is as follows:

Module 1. Exploring ASP.NET Core MVC

This module describes the Microsoft Web Technologies stack and compares the various ASP.NET programming models. It explores the structure of an MVC web application and identifies features such as models, views, controllers, configuration files, style sheets, and script files.

Module 2. Designing ASP.NET Core MVC Web Applications

This module explains how to plan the overall architecture of an ASP.NET Core MVC web application. It also explains how to plan the models, controllers, and views that are required to implement a given set of functional requirements.

Module 3. Configure Middleware and Services in ASP.NET Core

This module explains how to use existing middleware to set up an ASP.NET Core application and how to create your own middleware and use it to define custom behavior. The module also describes the basics of Dependency Injection and how it is used in ASP.NET Core. It also describes how to create a custom service, configure its scope, and inject it to middleware and ASP.NET Core MVC controllers.

Module 4. Developing Controllers

This module explains how to create controllers in an ASP.NET Core MVC web application. The module also describes how to write code in action filters that executes before or after a controller action and how to configure routes in an MVC application.

Module 5. Developing Views

This module explains how to create an MVC view and add Razor markup to it, use HTML helpers and tag helpers in a view, and reuse Razor markup in multiple locations throughout an MVC application.

Module 6. Developing Models

This module explains how to add a model to an MVC application, display the model data in a view by using helpers, and use forms with data annotations and validation.

Module 7. Using Entity Framework Core in ASP.NET Core

This module explains how to connect an application to a database to access and store data. It also explains what Entity Framework Core is and how to use Entity Framework Core to connect to a database from ASP.NET Core MVC applications.

Module 8. Using Layouts, CSS, and JavaScript in ASP.NET Core MVC

This module explains how to implement a consistent look and feel across an entire MVC web application. It also explains how to increase the responsiveness of the MVC web application by using JavaScript and jQuery.

Module 9. Client-Side Development

This module demonstrates how to use Bootstrap, Less, and Sass in an ASP.NET Core application. It also describes how to use task runners in an ASP.NET Core application and how to ensure that a web application displays correctly on devices with different screen sizes.

Module 10. Testing and Troubleshooting

This module explains how to unit test the components of an MVC application and to implement exception handling strategy for the application. It also describes how to log MVC applications.

Module 11. Managing Security

This module explains how to implement authentication and authorization for accessing an MVC web application. It also explains how to build an MVC web application that resists malicious attacks.

Module 12. Performance and Communication

This module explains how to improve the performance of ASP.NET Core applications by using caching and how to use state management technologies to improve the client experience by providing a consistent experience for the user. It also describes how to implement two-way communication by using SignalR.

Module 13. Implementing Web APIs

This module provides an overview of Web API and explains how to develop a Web API and call it from other applications.

Module 14. Hosting and Deployment

This module explains how to host and deploy a completed MVC application on Internet Information Services (IIS) and Microsoft Azure. It also describes how to use Azure to improve the capabilities of web applications.

Exam/Course Mapping

The following materials are included with your kit:

- **Course Handbook** is a succinct classroom learning guide that provides the critical technical information in a crisp, tightly focused format, which is essential for an effective in-class learning experience

You may be accessing either a printed course handbook or digital courseware material via the Skillpipe reader by Arvato. Your Microsoft Certified Trainer will provide specific details, but both printed and digital versions contain the following:

- Lessons guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
- Labs provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
- Module Reviews and Takeaways sections provide on-the-job reference material to boost knowledge and skills retention.
- Lab Answer Keys provide step-by-step lab solution guidance.

This course prepares student for the 70-486 exam.

To run the labs and demos in this course, use the code and instructions files that are available in GitHub:

- Instruction files: <https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/tree/master/Instructions>
- Code files: <https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/tree/master/Allfiles>

Make sure to clone the repository to your local machine. Cloning the repository before the course ensures that you have all the required files without depending on the connectivity in the classroom.

- Course evaluation. At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
 - To provide additional comments or feedback, or to report a problem with course resources, visit the Training Support site at <https://trainingsupport.microsoft.com/en-us>. To inquire about the Microsoft Certification Program, send an e-mail to certify@microsoft.com.

Module 1

Exploring ASP.NET Core MVC

Contents:

Module Overview	01-1
Lesson 1: Overview of Microsoft Web Technologies	01-3
Lesson 2: Overview of ASP.NET 4.x	01-12
Lesson 3: Introduction to ASP.NET Core MVC	01-22
Lab: Exploring ASP.NET Core MVC	01-29
Module Review and Takeaways	01-31

Module Overview

Microsoft ASP.NET Core MVC and the other web technologies of the ASP.NET Core can help you create and host dynamic, powerful, and extensible web applications. ASP.NET Core, of which ASP.NET Core MVC is part, is an open-source, cross-platform framework that allows you to build web applications. You can develop and run ASP.NET Core web applications on Windows, macOS, Linux, or any other platform that supports it.

ASP.NET Core MVC supports agile, test-driven development cycle. It also allows you to use the latest HTML standard and Front-End frameworks such as Angular, React, and more.

To build robust web applications, you need to be familiar with the technologies and products in the Microsoft web stack that we will cover in this module:

- You need to know how ASP.NET applications work with Web servers such as Kestrel and Internet Information Services (IIS).
- It is recommended that you be familiar with Visual Studio 2017 but note that with ASP.NET Core you can choose another Integrated Development Environment (IDE) such as Visual Studio Code.
- It is also recommended that you be familiar with Microsoft SQL Server, Entity Framework, Microsoft Azure, and Microsoft Azure SQL Database to deliver engaging web pages to visitors to the site.
- To choose a programming model that best suits a set of business requirements, you need to know how Model-View-Controller (MVC) applications differ from other ASP.NET programming models.

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

Objectives

After completing this module, you will be able to:

- Understand the variety of technologies available in the Microsoft web stack.
- Describe the different programming models available for developers in ASP.NET.
- Choose between ASP.NET Core and ASP.NET 4.x.

- Describe the role of ASP.NET Core MVC in the web technologies stack, and how to use ASP.NET Core MVC to build web applications.
- Distinguish between MVC models, MVC controllers, and MVC views.

Lesson 1

Overview of Microsoft Web Technologies

Before you use ASP.NET Core MVC, you need to know how it fits into ASP.NET Core in general, and where ASP.NET Core fits into the entire web technologies stack and particularly in the Microsoft web technologies stack. You should know how the ASP.NET Core websites are hosted, how they run server-side code on web servers and how the client-side code is run. You should also know which part of the code belongs to the server side and which part belongs to the client-side.

Lesson Objectives

After completing this lesson, you will be able to:

- Introduce Microsoft web technologies.
- Provide an overview of ASP.NET.
- Provide an overview of client-side web technologies, including AJAX and JavaScript libraries.
- Provide an overview of hosting technologies and Microsoft Azure particularly.

Introduction to Microsoft Web Technologies

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

Developer Tools

You can create simple websites with text and images by using a text editor, such as Notepad. However, most websites require complex actions to be performed on the server-side, such as database operations, email delivery, complex calculations, or graphics rendering. To create such complex, highly functional, and engaging websites quickly and easily, Microsoft provides several tools. The main ones are Microsoft Visual Studio 2017 and Microsoft Visual Studio Code.

Develop	Host	Execute	
		Server-Side	Client-Side
<ul style="list-style-type: none"> • Visual Studio • Visual Studio Code 	<ul style="list-style-type: none"> • IIS • Microsoft Azure 	<ul style="list-style-type: none"> • ASP.NET Core • ASP.NET 4.x 	<ul style="list-style-type: none"> • JavaScript • jQuery • Angular • React • AJAX

- **Visual Studio.** You can use Visual Studio 2017, an IDE, to create custom applications based on Microsoft technologies, regardless of whether these applications run on the web, on desktops, on mobile devices, or by using Microsoft cloud services. Visual Studio 2017 has rich facilities for designing, coding, and debugging any ASP.NET Core web application, including MVC applications.
- **Visual Studio Code.** It is a lightweight, free, and open-source code editor developed by Microsoft that is available for Windows, macOS, and Linux. Visual Studio Code has built-in support for JavaScript, HTML, cascading style sheets (CSS), Typescript, and Node.js and it has many extensions that allow you to create websites by using the language of your choice including C#, C/C++, Python, and more. It has a built-in integration with Git and powerful debugging tools for the web. You can use Visual Studio Code to create ASP.NET Core applications quickly by using the C# extension available on Visual Studio Code marketplace.

Hosting Technologies

Regardless of the tool you use to build a web application, you need a web server to host the web application. When users visit your site, the host server responds by rendering the HTML and returning it to the user's browser for display. The host server may query a database before it builds the HTML, and the host server may perform other actions such as sending emails or saving uploaded files. When you finish building the web application and make it ready for users to access on an intranet or over the internet, you must use a fully functional web server such as:

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

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

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

Code Execution Technologies

The code that you write in your code editor (or development environment) must run in one of two locations:

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

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

Server-Side Execution

Microsoft ASP.NET is a server-side web development framework that allows you to build server-based web applications. It was first introduced in 2002 with the first release of .NET Framework. Today you can choose between developing in ASP.NET 4.x and ASP.NET Core. ASP.NET 4.x is the latest version of the original ASP.NET. ASP.NET Core is a redesign of the original ASP.NET, and it is a much leaner and modular version.

You can write both ASP.NET 4.x and ASP.NET Core in Visual Studio 2017. The server-side code accesses the database, renders HTML pages, and returns them to the web browser. The MVC programming model is a part of both ASP.NET 4.x and ASP.NET Core. Other server-side technologies of choice include PHP and Node.js.

Client-Side Execution

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

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

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

One of the most popular types of AJAX-based web application is Single Page Applications (SPA). In those applications, parts of the page change dynamically based on the user's interaction with the page. There are many client-side frameworks and libraries that help developers to create such applications easily. Examples of SPA are the Angular framework and the React library. Both ASP.NET 4.x and ASP.NET Core integrate easily with these client-side SPA frameworks and in ASP.NET Core there are built-in templates for creating such applications.

Overview of ASP.NET

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

Programming Models

When you use ASP.NET to build an application, you are not restricted to a single style of programming; instead, you can choose from several different programming models. Some programming models are available only in ASP.NET 4.x, some are available only in ASP.NET Core and some are available in both ASP.NET 4.x and ASP.NET Core. Each programming model has a typical structure in the development environment and it stores code in specific places in the web hierarchy:

- **Web Pages.** Available only in ASP.NET 4.x.

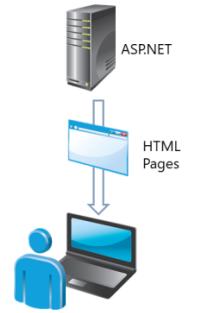
When you build a site by using Web Pages, you can write code by using the C# or Visual Basic programming language. If you write C# code, these pages have a .cshtml file extension. If you write Visual Basic code, these pages have a .vbhtml file extension. ASP.NET 4.x runs the code in these pages on the server to render data from a database, respond to a form post, or perform other actions. This programming model is simple and easy to learn and is suited for simple data-driven sites.

- **Web Forms.** Available only in ASP.NET 4.x.

When you build a site by using Web Forms, you employ a programming model with rich server-side controls and a page life cycle that is not unlike building desktop applications. Built-in controls include buttons, text boxes, and grid views for displaying tabulated data. You can also add third-party

Programming Models:

- ASP.NET 4.x:
 - Web Pages
 - Web Forms
- ASP.NET 4.x and ASP.NET Core:
 - MVC
 - Web API
- ASP.NET Core:
 - Razor Pages



controls or build custom controls. To respond to user actions, you can attach event handlers containing code to the server-side controls. For example, to respond to a click on a button called **firstButton**, you could write code in the **firstButton_Click()** event handler.

- **MVC**. Available in both ASP.NET Core and ASP.NET 4.x.

When you build a site by using MVC, you separate server-side code into three parts:

- **Model**. An MVC model defines a set of classes that represent the object types that the web application manages. For example, the model for an e-commerce site might include a Product model class that defines properties such as Description, Catalog Number, Price, and others. Models often include data access logic that reads data from a database and writes data to that database.
- **Controllers**. An MVC controller is a class that handles user interaction, creates and modifies model classes, and selects appropriate views. For example, when a user requests full details about a particular product, the controller creates a new instance of the Product model class and passes it to the Details view, which displays it to the user.
- **Views**. An MVC view is a component that builds the web pages that make up the web application's user interface. Controllers often pass an instance of a model class to a view. The view displays properties of the model class. For example, if the controller passes a Product object, the view might display the name of the product, a picture, and the price.

This separation of model, view, and controller code ensures that MVC applications have a logical structure, even for the most complex sites. It also improves the testability of the application.

Ultimately, MVC enables more control over the generated HTML than either Web Pages or Web Forms.

- **Razor Pages**: Available only in ASP.NET Core.

This programming model was first introduced with ASP.NET Core. While in MVC we have a controller that receives requests, a model that represents the data and a view that renders the HTML, with Razor Pages this is different. The request goes straight to a page that is generally located in the Pages folder. Those Razor Pages have a .cshtml extension and they can build the model and render an HTML in a single file while still having a separation of concerns within them. Each Razor Page has an accompanying .cs file that contains all the methods, model handlers, and logic.

- **Web API**: Available in both ASP.NET Core and ASP.NET 4.x.

When you build a website using Web API, it is possible to separate your client-side code from your server-side code completely and to use AJAX to create client-server communication. Web API can be accessed via the web by using HTTP requests. Web API is very useful when you want to expose parts of your application functionality, data or services and when you want to create SPA applications and other applications by using the Representational State Transfer (REST) API. As with the MVC programming model, the code is also separated into distinct parts:

- **Controller**. A Web API Controller is a class that handles the client request that was sent to the server. It accesses the database, retrieves information, updates the database, if needed, and returns the HTTP response including a status code that indicates whether the action succeeded and the data, if needed.
- **Model**. As With MVC, it is a set of classes that represent the object types that the web application manages.
- **Client**. The client sends requests to the server to run specific actions in the Web API Controller. On the server side, you create an interface that consists of functions that can be accessed via HTTP. Those calls are sent from the client to the server to retrieve specific information and perform read-write operations.

As the name Web API suggests, the API exists over the web and can be accessed by using HTTP protocol.

Client-Side Web Technologies

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

- JavaScript
- jQuery
- AJAX
- Angular
- React
- And more



JavaScript

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

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

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

Different browsers interpret JavaScript differently. When you develop your website, you don't know what browsers your visitors will use. Therefore, you must write JavaScript that works around browser compatibility.

jQuery

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



Note: There are many other JavaScript libraries and frameworks such as Underscore, D3, Angular, and React. If you find any of these better suited for a particular task, or if you have experience in developing web applications by using them, you can include them in your ASP.NET pages, instead of jQuery or together with it.

AJAX

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

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

Angular

Angular is a front-end framework that allows you to build robust web applications and solve common development challenges easily. Modern web applications require us to implement complex scenarios such as data-binding, routing, and internationalization. Angular enables you to do those and keep your code maintainable, testable, and corresponding to the latest coding best practices.

Angular is written entirely in TypeScript. TypeScript is a superset of JavaScript that was created by Microsoft allowing you to use the latest JavaScript standards and features without worrying about backward compatibility. TypeScript is not processed by browsers and it needs to be transformed into JavaScript to allow browsers to run your code. This transformation is called Transpiling and it happens during the build process. Its output is pure JavaScript code. Typescript also has a static optional type system and it allows you to write object-oriented code.

Angular is also called a client-side Component-Based framework which means that the code you write is separated into small and functional pieces of code. Angular has a built-in separation of concerns and its architecture is similar to MVC. It has dedicated code parts for models, controllers, and views. JavaScript objects serve as models, components serve as controllers, and HTML templates serve as views.

React

React is a JavaScript library that is dedicated to building user interfaces. It focuses on making interactive UIs and designing views while speeding up the development cycle. It is a Component-Based library where each component includes its own logic and maintains its own state. React renders views quickly and focuses on high performing UIs. React Native is a framework that is based on React and JavaScript and allows developers to create high performing mobile applications.

Hosting Technologies

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

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

- IIS
 - Features
 - Scaling
 - Perimeter Networks
- IIS Express
- Other Web Servers



Internet Information Services Features

IIS is tightly integrated with ASP.NET, Visual Studio 2017, and Windows Server. It includes the following features:

- **Deployment Protocols.** The advanced Web Deploy protocol, which is built into Visual Studio 2017, automatically manages the deployment of a website with all its dependencies. Alternatively, you can use File Transfer Protocol (FTP) to deploy content.
- **Centralized Web Farm Management.** When you run a large website, you can configure a load-balanced farm of many IIS servers to scale to large sizes. IIS management tools make it easy to deploy sites to all servers in the farm and manage sites after deployment.
- **High-Performance Caches.** You can configure ASP.NET to make optimal use of the IIS caches to accelerate responses to user requests. When IIS serves a page or other content, it can cache it in memory so that subsequent identical requests can be served faster.
- **Authentication and Security.** IIS supports most common standards for authentication, including Smart Card authentication and Integrated Windows authentication. You can also use Transport Layer Security (TLS) to encrypt sensitive communications, such as login pages and pages containing credit card numbers.
- **ASP.NET Support.** IIS is a web server that fully supports ASP.NET.
- **Other Server-Side Technologies.** You can host websites developed in PHP and Node.js on IIS.

Scaling Up IIS

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

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

Perimeter Networks

Web servers, including IIS, are often located on a perimeter network. A perimeter network has a network segment that is protected from the internet through a firewall that validates and permits incoming HTTP requests. A second firewall, which permits requests only from the web server, separates the perimeter

network from the internal organizational network. Supporting servers, such as database servers, are usually located on the internal organizational network.

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

IIS Express

Internet Information Services Express is a lightweight version of IIS, which is optimized for development and testing. IIS Express does not provide all the features of IIS on Windows Server. For example, you cannot create load-balanced server farms by using IIS Express. However, it has all the features necessary to host rich ASP.NET websites and other websites on a single server. It also does not require administrator rights to host websites, which means multiple developers can use the same server to host websites without full control over the host machine.

Other Web Servers

Apache is a popular non-Microsoft web server and there are other alternatives such as Nginx. Both can be installed on Windows Server or Windows client computers to host websites during development or for production deployments.

Microsoft Azure

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

- **Flexible scaling.** As the needs of your web application and database grow, extra server resources are automatically allocated. You don't need to plan for server farms or load balancing systems because these are built into Azure. Scaling is instantaneous: it doesn't require purchasing hardware, installing it in racks, and configuring its software. With the correct architecture, your application can scale indefinitely as demand increases.
- **Flexible pricing.** With Azure, you can choose a pay-as-you-go pricing model, which means that you only pay for the data that you use. This makes Azure very cost-efficient for small websites. It also makes costs predictable for large websites.
- **Worldwide presence.** Azure datacenters are spread around the world, and you can easily replicate your application in multiple datacenters to bring it geographically closer to your customers. This also improves your application's availability in case of a catastrophic event that brings down an entire datacenter in one region, because the application can automatically fail-over to another region.
- **First-class security and reliability.** Azure datacenters, computers, disks, networking devices, and other facilities are managed with world-class security and reliability in mind. Hardware failures are completely masked by having multiple redundant backups, which ensure your application's reliability.

Some of the application and service types that you can host on Azure include the following:

- **Web Apps.** You can host an entire website on Azure. Azure supports websites developed in ASP.NET, PHP, Java, Ruby, Python, and Node.js. You can also deploy websites to Azure directly from Visual Studio 2017. The Web Apps feature of Azure App Service does not have to contain a user interface; it can be an HTTP API designed to be called from other software or services, such as a mobile device or a desktop application.
- **Databases.** When you use IIS to host a website, you can use SQL Server to store the website database. When you host a website in Azure, you can use Azure SQL Database, which is a cloud-based database service based on SQL Server, to host your database. Azure also offers Cosmos DB, which is a hybrid database that can seamlessly scale across multiple datacenters around the world.

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

Lesson 2

Overview of ASP.NET 4.x

ASP.NET 4.x is a web framework that lets us build rich websites and web applications. ASP.NET 4.x helps developers to create dynamic websites that use client-side and server-side code. In addition, with ASP.NET 4.x, you are not restricted to a single style of programming; instead, you can choose from several different programming models: Web Pages, Web Forms, MVC, and Web API. These programming models differ from each other, and they have their own advantages and disadvantages in different scenarios. ASP.NET 4.x also provides many features that you can use, regardless of the programming model you choose.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Web Pages programming model.
- Describe the Web Forms programming model.
- Describe the MVC programming model.
- Describe the Web API programming model.
- Determine whether to build Web Pages, Web Forms, MVC, or Web API web applications, based on customer needs.
- Describe the features that can be used in all ASP.NET 4.x applications, regardless of the chosen programming model.

Overview of Web Pages

Web Pages is the simplest programming model you can choose to build a web application in ASP.NET 4.x. You can use Web Pages to create a website quickly and with little technical knowledge. There is a single file associated with each webpage that users can request. For each page, you write HTML, client-side code, and server-side code in the same .cshtml file. The @ symbol is used to distinguish server-side code from HTML and JavaScript. When users request the page, the ASP.NET 4.x runtime compiles and runs the server-side code to render HTML and returns that HTML to the web browser for display.

- Code in .CSHTML files
- Precise control over HTML

```
<h2>Special Offers</h2>
<p>Get the best possible value on Northwind specialty
 foods by taking advantage of these offers:</p>
@foreach (var item in offers){
<div class="offer-card">
<div class="offer-picture">
@if (!String.IsNullOrEmpty(item.PhotoUrl)){
    
}
</div>
</div>
}
```



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

You can use Visual Studio 2017 to create Web Pages applications.

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

A Web Pages code example

```
<h2>Special Offers</h2>
<p>Get the best possible value on Northwind specialty foods by taking advantage of these offers:</p>
@foreach (var item in offers) {
    <div class="offer-card">
        <div class="offer-picture">
            @if (!String.IsNullOrEmpty(item.PhotoUrl)){
                
            }
        </div>
    </div>
}
```

The Web Pages programming model has the following advantages:

- It is simple to learn.
- It provides precise control over the rendered HTML.

Using a Web Pages site has some disadvantages:

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

Overview of Web Forms

Web Forms is another programming model that you can choose in ASP.NET 4.x. A Web Forms application is characterized by controls, which you can drag from the Visual Studio toolbox onto each webpage. This method of creating a user interface resembles the method used in desktop applications.

- Code in .aspx files and code-behind files
- Create a UI by dragging controls onto a page
- Controls provide rich properties and events
- Bind controls to data

Web Forms Controls

ASP.NET 4.x provides a wide variety of highly-functional controls that you assemble on Web Forms. After you add a control to a page, you can write code to respond to user events. For example, you can use the code in a button click event to process a user's input in a form. The controls provided include:

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

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

Web Forms Code Files

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

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

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

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

Binding Controls to Data

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

Advantages and Disadvantages of Web Forms

The Web Forms programming model has the following advantages:

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

Using a Web Forms site has some disadvantages:

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

Overview of MVC

MVC is another programming model available in ASP.NET 4.x. MVC applications are characterized by a strong separation of business logic, data access code, and the user interface into models, controllers, and views. It is a more advanced programming model than Web Forms and Web Pages and is suitable for creating large-scale applications.

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

Models

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

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

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

Views

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

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

Controllers

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

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

Advantages and Disadvantages of MVC

The MVC programming model has the following advantages:

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

Using an MVC site has some disadvantages:

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

Discussion: ASP.NET 4.x Application Scenarios

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

Database Front-End

Your organization has its own customer relationship management system that stores data in a SQL Server database. Your team of developers wrote the user interface in Visual Studio 2017 as a desktop application. The directors now require that all computers should be able to access the application even when the desktop client application is not installed. Because all computers have a browser, you have decided to write a web application in ASP.NET 4.x to enable this.

Which programming model will you use in the following scenarios? (Choose one of the following in each scenario: MVC, Web Forms, Web Pages)

- A database front-end to be hosted on an intranet
- An e-commerce site for a large software organization
- A website for a small charitable trust

E-Commerce Site

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

Website for a Small Charitable Trust

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

Shared ASP.NET 4.x Features

ASP.NET 4.x also includes a range of shared features that are available regardless of the programming model that you use. This means that if you are familiar with these features from working with one programming model, your knowledge can be used also in another programming model.

- Configuration
- Authentication
- Membership and Roles
- State Management
- Caching

The ASP.NET 4.x API

Whichever programming model you choose, you have access to the classes from the ASP.NET 4.x API. These classes are included in .NET Framework in namespaces within the **System.Web** namespace and can be used to rapidly implement common website functionalities such as:

- **Configuration.** Using the Web.config files, you can configure your web application, regardless of the programming model. The Web.config files are XML files with specific tags and attributes that the ASP.NET 4.x runtime accepts. For example, you can configure database connections and custom error pages in the Web.config file. In code, you can access the configuration through the **System.Web.Configuration** namespace.

- **Authentication and Authorization.** Many websites require users to log on by entering a username and password, or by providing extra information. You can use ASP.NET 4.x membership providers to authenticate and authorize users and restrict access to content. You can also build pages that enable users to register a new account, reset a password, recover a lost password, or perform other account management tasks. Membership providers belong to the **System.Web.Security** namespace.
- **Caching.** ASP.NET 4.x may take some time to render a complex webpage that may require multiple database queries or calls to external services. You can use caching to mitigate this delay. ASP.NET 4.x caches a rendered page in memory so that it can return the same page to subsequent user requests without having to render it again from the start. In a similar manner, .NET Framework objects can also be cached. You can access cached pages by using the **System.Runtime.Caching** namespace and configure the caches in Web.config.

Compiling ASP.NET Code

Because ASP.NET 4.x server-side code uses .NET Framework, you must write code in a .NET managed programming language such as C# or Visual Basic. Before running the code, it must be compiled into native code so that the server CPU can process it. This is a two-stage process:

1. **Compilation to MSIL.** When you build a website in Visual Studio, the ASP.NET 4.x compiler creates .dll files with all the code compiled into Microsoft Intermediate Language (MSIL). This code is both independent of the language you used to write the application and the CPU architecture of the server.
2. **Compilation to native code.** When a page is requested for the first time, the Common Language Runtime compiles MSIL into native code for the server CPU.

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

When you use the default compilation model, delays can arise when the first user requests a page. This is because ASP.NET 4.x must compile the page before serving it to the browser. To avoid such delays and to protect source code, use pre-compilation. When you pre-compile a site, all the ASP.NET 4.x files, including controllers, views, and models, are compiled into a single .dll file.



Additional Reading: For more information about ASP.NET 4.x compilation, see:
<https://aka.ms/moc-20486d-m1-pg4>

Configuration

When you configure an ASP.NET 4.x site, you can control how errors are handled, how the site connects to databases, how user input is validated, and many other settings. You can configure ASP.NET 4.x sites by creating and editing the Web.config files. The Web.config file in the root folder of your site configures the entire site, but you can override this configuration at lower levels by creating Web.config files in sub-folders.

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

An example of a Web.config file

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />
<add key="webpages:Version" value="3.0.0.0" />
<add key="webpages:Enabled" value="false"/>
<add key="ClientValidationEnabled" value="true"/>
<add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
<system.web>
<compilation debug="true" targetFramework="4.6.1"/>
<httpRuntime targetFramework="4.6.1"/>
</system.web>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<dependentAssembly>
<assemblyIdentity name="System.Web.Helpers" publicKeyToken="31bf3856ad364e35"/>
<bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0"/>
</dependentAssembly>
<dependentAssembly>
<assemblyIdentity name="System.Web.WebPages" publicKeyToken="31bf3856ad364e35"/>
<bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0"/>
</dependentAssembly>
<dependentAssembly>
<assemblyIdentity name="System.Web.Mvc" publicKeyToken="31bf3856ad364e35"/>
<bindingRedirect oldVersion="1.0.0.0-5.2.3.0" newVersion="5.2.3.0"/>
</dependentAssembly>
</assemblyBinding>
</runtime>
<system.codedom>
<compilers>
<compiler language="c#;cs;csharp" extension=".cs"
type="Microsoft.CodeDom.Providers.DotNetCompilerPlatform.CSharpCodeProvider,
Microsoft.CodeDom.Providers.DotNetCompilerPlatform, Version=1.0.8.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"
warningLevel="4" compilerOptions="/langversion:default /nowarn:1659;1699;1701"/>
<compiler language="vb;vbs;visualbasic;vbscript" extension=".vb"
type="Microsoft.CodeDom.Providers.DotNetCompilerPlatform.VBCodeProvider,
Microsoft.CodeDom.Providers.DotNetCompilerPlatform, Version=1.0.8.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"
warningLevel="4" compilerOptions="/langversion:default /nowarn:41008
#define:_MYTYPE="Web" /optionInfer+/">
</compilers>
</system.codedom>
</configuration>
```

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

Authentication

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

ASP.NET 4.x supports several mechanisms for authentication. For example, if you are using Microsoft Edge on a Windows computer, ASP.NET 4.x may be able to use Integrated Windows authentication. In this mechanism, your Windows user account is used to identify you. When you build internet sites, you cannot be sure that users have Windows, a compatible browser, or an accessible account, so Forms Authentication is often used. Forms Authentication is supported by many browsers and it can be configured to check credentials against a database, directory service, or other user account stores.

Membership and Roles

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

In ASP.NET 4.x, a membership provider is a component that implements user account management features. Several membership providers are supported by ASP.NET 4.x, such as the SQL membership provider, which uses a SQL database to store user accounts. You can also create a custom membership provider, inheriting from one of the default providers if you have unique requirements.

When you have more than a few users, you may want to group them into roles with different levels of access. For example, you might create a **Gold Members** role containing user accounts with access to the best special offers. ASP.NET 4.x role providers enable you to create and populate roles with the minimum of custom code.

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

State Management

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

However, when you build a web application, you must frequently preserve values across multiple page requests. For example, if a user places a product in a shopping cart, and then clicks **Check Out**, this is a separate web request, but the server must preserve the contents of that shopping cart; otherwise, the shopping cart will be emptied and the customer will buy nothing. ASP.NET 4.x provides several locations where you can store such values or state information across multiple requests.

Caching

An ASP.NET 4.x page is built dynamically by the ASP.NET 4.x runtime on the web server. For example, in a Web Pages application, the runtime must run the C# code in the page to render HTML to return it to the browser. That C# code may perform complex and time-consuming operations. It may run multiple queries against a database or call services on remote servers. You can mitigate these time delays by using ASP.NET 4.x caches.

For example, you can use the ASP.NET 4.x page cache to store the rendered version of a commonly requested page in the memory of the web server. The front page of your product catalog may be requested hundreds or thousands of times a day by many users. If you cache the page in memory the first time it is rendered, the web server can serve it to most users very rapidly, without querying the database server and building the page from scratch.

Overview of Web API

Web API is another programming model that you can choose in ASP.NET 4.x. API stands for Application Programming Interface. It is an interface that exposes some functions and allows programmers to access specific sets of data, features, and business logic in your application. Web API is an API that exists over the web and can be accessed using the HTTP protocol.

Web API also allows you to implement REST services in your application and with that reduce application overhead and limit the data that is transmitted between client and server systems.

Web API allows you to call methods it exposes by using the server-side or client-side code and it allows you to implement REST-style Web APIs in your application.

- Helps creating RESTful APIs
- Enables external systems to use your application's business logic
- Accessible to various HTTP clients
- Helps to obtain data in different formats such as JSON and XML
- It supports create, read, update and delete (CRUD) actions
- Ideal for mobile application integration



Call a Web API using AJAX

Web API allows a web client to call methods that it exposes by using AJAX. There are many technologies that can be used to perform AJAX calls which include:

- **JavaScript**. Allows you to perform AJAX calls using the **XMLHttpRequest** object or the **fetch** method.
- **jQuery**. Simplifies the process of writing AJAX calls by using the **\$.ajax** method or other methods available in jQuery such as **\$.get** or **\$.post**.
- **Angular**. Has dedicated objects for performing AJAX calls.

And there are many more.

 **Note:** RESTful API, in general, is a programming model that can be implemented in many programming languages and frameworks. You can choose to implement RESTful APIs in various technologies such as Java, PHP, and Node.js, but Web API is the one to choose when working with ASP.NET 4.x.

Advantages and Disadvantages of Web API

The Web API programming model has the following advantages:

- Helps in creating RESTful APIs.
- Enables external systems to use your application's business logic and features.
- Can be accessed by various HTTP clients such as Windows, Android, IOs, and more.
- Helps obtain data in different formats such as JSON, XML, and custom formats.
- Supports create, read, update and delete (CRUD) actions since it works with HTTP verbs such as GET, POST, PUT, and DELETE.
- Is ideal for mobile application integration.

Using Web API has some disadvantages:

- With Web API you have a complete separation between server-side code and client-side code. For some programmers, this is an advantage, but others may find this challenging and hard to understand.

- The data returned from Web API usually is not indexed by search engines such as Google, more work needs to be done to make it crawlable.

Lesson 3

Introduction to ASP.NET Core MVC

In this lesson, you are going to learn how models, views, and controllers work together to render HTML. You will see the structure of MVC applications and how it determines the display of information in a Visual Studio 2017 project. Also, you will examine various features of MVC projects, included in ASP.NET Core, that help developers build rich and engaging applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe models, views, and controllers.
- Choose between ASP.NET 4.x and ASP.NET Core.
- Choose between .NET Core and .NET Framework.
- Describe the key features of an ASP.NET Core MVC web application in Visual Studio 2017.

Introduction to ASP.NET Core

ASP.NET Core is a cross-platform, open-source framework that allows you to create and host dynamic, powerful, and extensible web-based applications. With ASP.NET Core you can develop on various platforms – Windows, macOS, and Linux, deploy your application to the cloud, and create services and mobile backends.

- Introduction to ASP.NET Core
- Discussion: Choose between ASP.NET 4.x and ASP.NET Core
- Choose between .NET Core and .NET Framework
- Models, Views, and Controllers
- Demonstration: How to Explore an ASP.NET Core MVC Application

 **Note:** ASP.NET Core can target .NET Core or .NET Framework. Note that the difference between the two will be covered later in this lesson, in the topic "Choose between .NET Core and .NET Framework".

Why should you use ASP.NET Core?

ASP.NET Core is a redesign of ASP.NET 4.x and it brings in many architectural shifts. ASP.NET Core is much leaner and modular than ASP.NET 4.x and has many changes within it.

ASP.NET Core includes three main programming models that can help you to develop your applications smoothly: MVC, Razor Pages, and Web API.

MVC

When you build a website using ASP.NET Core MVC, you separate your code into three parts: model, view, and controller. This separation of model, view, and controller code ensures that MVC applications have a logical structure, even for the most complex sites. It also improves the testability of the application.

 **Note:** MVC will be covered in greater depth in the rest of the modules in this course.

Razor Pages

This programming model was first introduced with ASP.NET Core and it is an alternative to the MVC programming model. While in MVC we have a controller that receives requests, a model and a view that generates the response, with Razor Pages it is different. The request goes straight to a page that is generally located in the Pages folder.

Each Razor Page file has an accompanying **.cshtml.cs** file that contains all the methods, model handlers, and logic. It contains a class that inherits from **PageModel** that will initialize the model inside the **OnGet** method. The **OnGet** method is also the place to get data from the database. Each public property inside the model can be displayed on the page using Razor syntax.

This code example shows what the code-behind Razor Page looks like:

A Razor Pages code example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPagesExample.Pages
{
    public class HomePageModel : PageModel
    {
        public string Description { get; set; }
        public string MoreInfo { get; set; }

        public void OnGet()
        {
            Description = "Your application description is here.";
            MoreInfo = "Your application extra information is here.";
        }
    }
}
```

Razor Page syntax looks a lot like a Razor view, which will be covered in Module 5 "Developing Views". What makes it different is the **@page** directive that must be at the first line of the .cshtml file. **@page** converts the file into a Razor Page, which means that it handles requests directly without going through a controller like in the MVC programming model.

Razor syntax can be used inside Razor Pages. It allows you to insert the properties from the model and other sources into the HTML markup. Razor syntax starts with @ character. When you want to display properties from the model inside the HTML markup, you will use **@model**. **@model** will have a reference to the properties you initialized inside the **OnGet** method.

This code example shows how to edit Razor Pages and display values from the model inside the page:

Using Razor syntax inside Razor Pages

```
@page
@model HomePageModel
 @{
     ViewData["Title"] = "Home page";
 }

<div class="row">
    <div class="col-md-3">
        <h1>@ViewData["Title"] </h1>
        <ul>
            <li>@model.Description</li>
            <li>@model.MoreInfo</li>
        </ul>
    </div>
</div>
```

```
</ul>
</div>
</div>
```

Web API

Web API is very useful when creating SPA applications and other applications that use REST API. As with the MVC programming model, the code is also separated into distinct parts:

- **Controller.** A Web API controller is a class that handles the client request that was sent to the server. It accesses the database, retrieves information, updates the database, if needed, and returns the HTTP response, including status code that indicates whether the action succeeded and data, if needed.
- **Model.** As with MVC, it is a set of classes that represent the object types that the web application manages.
- **Client.** The client sends requests to the server to run specific actions in the Web API controller. On the server side, there is an interface that consists of functions that can be accessed via HTTP. Those calls are sent from the client to the server to retrieve specific information and perform read-write operations.

As the name Web API suggests, the API exists over the web and can be accessed by using HTTP protocol.



Note: Web API will be covered in Module 13, "Implementing Web APIs".



Note: Note that Web Forms and Web Pages don't exist in ASP.NET Core. If you want to use those programming models, you should use ASP.NET 4.x instead.

Discussion: Choose between ASP.NET 4.x and ASP.NET Core

When developing a web application with ASP.NET, you have to choose between using ASP.NET 4.x or ASP.NET Core. The following scenarios describe some requirements for web applications. For each scenario, describe what technology you would prefer – ASP.NET 4.x or ASP.NET Core and what are the reasons for your choice.

High Performing Application for the Stock Market

You want to develop a web application that shows the latest trends in the economic market.

Your application must have high performance and must be very fast at fetching the latest information from various sources.

Which version of ASP.NET (ASP.NET 4.x or ASP.NET Core) will you use in the following scenarios?

- High performing application for the stock market
- Update an existing website written in Web Forms
- Open-source project running on macOS

Update an Existing Website Written in Web Forms

You are working in a large organization that has been using Web Forms for many years. They have an existing website written in Web Forms. You are asked to extend its functionality and add few pages.

Open Source Project running on macOS

You have an idea for a brand-new content management system. You want it to be an open-source project and develop it on your home Mac computer running macOS. You choose to use Visual Studio Code as your IDE.

Choose between .NET Core and .NET Framework

When building server-side applications with .NET, you must choose between its two main implementations: .NET Framework and .NET Core. They might seem to be similar and you can share code across the two thanks to .NET Standard that will be discussed below, but there are meaningful differences between them. Your preference should be based on your goals and what you are trying to accomplish with your application.

.NET Standard

.NET Standard is a formal specification of the .NET APIs. It is intended to be available on all .NET implementations and its goal is to create a unification of the .NET ecosystem. It enables you to create code that is portable across all .NET implementations. Each .NET implementation is targeting a specific version of .NET Standard.

A higher version number means that the previous versions are also supported but it has some extensions. For example, .NET Framework 4.6.2 implements .NET Standard 1.5. It means that it exposes all APIs defined in .NET Standard versions 1.0 through 1.5. Similarly, .NET Core 2.0 implements .NET Standard 2.0 and for that reason it exposes all APIs defined in .NET Standard versions 1.0 through 2.0. This also means that .NET Core 2.0 supports the code written in .NET Framework 4.6.2 because it implements .NET Standard 1.5.

When to use .NET Core for your application?

The following cases describe when you should choose .NET Core over .NET Framework for your application.

You want your code to work cross-platform

If your application needs to run on multiple platforms such as Windows, Linux, and macOS, .NET Core should be used because it supports those three operating systems. With .NET Core, you can use Windows, Linux, and macOS for your development environment.

Visual Studio 2017 or Visual Studio Code could be used as the IDE. Many third-party editors such as Sublime Text work with .NET Core. You can also avoid using code editors completely and use .NET Core CLI tools instead. It is available for all supported platforms.

You want to create microservices

.NET Core was built with microservice architecture in mind. In .NET Framework, the services layer contains all the functionality of the application with references to libraries and main components. In microservices architecture, the services layer is broken down into multiple independent components. A microservices architecture also allows you to create a mix of technologies across a service boundary. For example, combine services created in Java and .NET Framework with new ones created in .NET Core. This allows you to gradually adopt .NET Core and use it for new microservices while still maintaining the old ones.

You want to use Docker containers

Containers are frequently used along with a microservices architecture, but they can also be used to containerize web applications or services following any architectural pattern. While .NET Framework can be used on Windows containers, .NET Core can be deployed also on Linux Docker containers. This is thanks to the cross-platform nature of .NET Core. When creating and deploying a container, the size of the .NET Core image is smaller than with .NET Framework. That together makes .NET Core a better choice for containers.

- You should use .NET Core when:
 - You want your code to run cross-platform
 - You want to create microservices
 - You want to use Docker containers
 - You want to achieve a high-performing scalable system
- You should use .NET Framework when:
 - You want to extend an existing application that uses .NET Framework
 - You want to use NuGet packages or third-party .NET libraries that are not supported in .NET Core
 - You want to use .NET technologies that aren't supported in .NET Core
 - You want to use a platform that doesn't support .NET Core

You want to achieve a high-performing scalable system

When you want to achieve the best possible performance and scalability, .NET Core is the better choice. This is extremely important in microservice architecture as it needs a lower number of virtual machines to run. This makes .NET Core a top performing web framework according to many sources today.

When to use .NET Framework for your application

The following cases describe when you should choose .NET Framework over .NET Core for your application.

You want to extend an existing application that uses .NET Framework.

If you have an existing application written in .NET Framework, it is not recommended to migrate it to .NET Core. Instead, you should use .NET Core to extend your application. For example, you can write a new Web API in ASP.NET Core.

You want to use NuGet packages or third-party .NET libraries that are not supported in .NET Core.

While many third-party libraries adopted .NET Core, there might be libraries and NuGet packages that are not being updated or that do not support the .NET Standard that enables sharing code between all .NET implementations. In that case, it is better to use .NET Framework.

You want to use .NET technologies that aren't supported in .NET Core

Some of the .NET Framework technologies aren't available in .NET Core. While some of them might be available in the future, others are not compatible with the design patterns implemented in .NET Core and will not be added. For example, ASP.NET Web Forms and ASP.NET Web Pages are only available in .NET Framework. If you want to use any of those, you should choose .NET Framework over .NET Core.

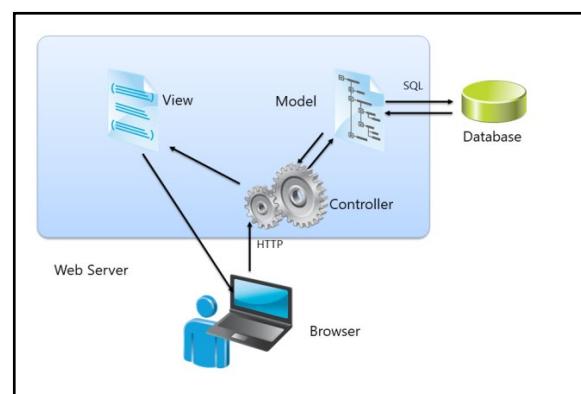
You want to use a platform that doesn't support .NET Core.

There are some platforms that don't support .NET Core but support .NET Framework. If you are using a platform that does not support .NET Core and you don't want to change the platform, choose .NET Framework instead of .NET Core.

Models, Views, and Controllers

When developing ASP.NET Core MVC applications, you need to distinguish between models, views, and controllers. You need to understand the unique role each one of them plays and how they work together to create web applications.

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



Models and Data

A model is a .NET class that represents an object handled by your website. For example, the model for an e-commerce application may include a Product model class with properties such as Product ID, Part Number, Catalog Number, Name, Description, and Price.

Like any other .NET classes, model classes can include a constructor, which is a procedure that runs when a new instance of that class is created. You can also include other procedures, if necessary. These procedures encapsulate the business logic. For example, you can write a Publish procedure that marks the product as ready-to-sell.

Most websites store information in a database. In an MVC application, the model includes code that reads and writes database records. ASP.NET Core MVC works with many different data access frameworks. A commonly used framework is Entity Framework Core, which will be covered in Module 7, “Using Entity Framework Core in ASP.NET Core”.

Controllers and Actions

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

Controllers inherit from the **Microsoft.AspNetCore.Mvc.Controller** base class. Actions usually return an object that implements the **Microsoft.AspNetCore.Mvc.IActionResult** interface.

Views and Razor

A view is a .cshtml file that includes both the HTML markup and programming code. A view engine interprets the view files, runs the server-side code, and renders HTML to the web browser. Razor is the default view engine in ASP.NET Core MVC.

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

Part of a Razor view

```
<h2>Details</h2>

<fieldset>
<legend>Comment</legend>

<div class="display-label">
    @Html.DisplayNameFor(model => model.Subject)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.Subject)
</div>

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

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

Request Life Cycle

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

1. The user requests the web address: <http://www.adventure-works.com/product/display/1>.

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

Demonstration: How to Explore an ASP.NET Core MVC Application

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

Demonstration Steps

You will find the steps in the "Demonstration: How to Explore an ASP.NET Core MVC Application" section on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD01_DEMO.md.

Lab: Exploring ASP.NET Core MVC

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 ASP.NET Core MVC application for your organization's customers, similar to the 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 their sales. You have been asked to begin the planning of the application. You have also been asked to examine programming models available to ASP.NET Core developers. To do this, you need to create basic web applications using three different models: Razor Pages, Web API, and MVC.

Objectives

After completing this lab, you will be able to:

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

Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD01_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD01_LAK.md.

Exercise 1: Exploring a Razor Pages Application

Scenario

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

The main tasks for this exercise are as follows:

1. Creating a Razor Pages application.
2. Exploring the application structure.
3. Adding simple functionality.
4. Running the application.

Exercise 2: Exploring a Web API Application

Scenario

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

The main tasks for this exercise are as follows:

1. Creating a Web API application.
2. Exploring the application structure.
3. Adding simple functionality.

4. Running the application.

Exercise 3: 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. Creating an MVC application.
2. Explore the application structure.
3. Add simple functionality.
4. Run the application.

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

Question: A member of your team replaced the line `return View(model);` in the **Details** action of the **AnimalController** class with the line `return View();`. What will happen when the **Details** action is called?

Question: When you run the **CakeStoreApi** application, the browser displays **value1** and **value2**. You want to display your first name and your last name instead. What will you have to do to achieve your goal?

Module Review and Takeaways

In this module, you have seen the tools, technologies, and web servers that are available in the Microsoft web stack that you can use to build and host web applications on the internet. You should also be able to distinguish between applications written in the three ASP.NET Core programming models: Razor Pages, Web API, and MVC. In this module, we also discussed the differences between ASP.NET 4.x and ASP.NET Core and you should be able to know which to use in various scenarios. Finally, you should be able to use MVC applications to render pages by using models, views, and controllers.

Review Questions

Question: What is the main difference between Razor Pages and MVC?

Question: Which of the application programming models will you recommend for the organization's customers: Razor Pages, Web API, or MVC?

Best Practices

- Use Web API when you want to create HTTP services and to allow developers access specific sets of data or functionality of your application. If you would like to add a UI, you should combine Web API with another programming model that has UI functionality.
- 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 TDD.

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: <https://aka.ms/moc-20486d-m1-pg5>

Module 2

Designing ASP.NET Core MVC Web Applications

Contents:

Module Overview	02-1
Lesson 1: Planning in the Project Design Phase	02-2
Lesson 2: Designing Models, Controllers and Views	02-15
Lab: Designing ASP.NET Core MVC Web Applications	02-22
Module Review and Takeaways	02-24

Module Overview

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

Objectives

After completing this module, you will be able to:

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

Lesson 1

Planning in the Project Design Phase

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

Lesson Objectives

After completing this lesson, you will be able to:

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

Project Development Methodologies

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

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

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

Waterfall Model

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

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

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

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

Iterative Development Model

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

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

Prototyping Model

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

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

Agile Software Development Model

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

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

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

Extreme Programming

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

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

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

 **Additional Reading:** For more information about the extreme programming model, refer to "Extreme Programming: A gentle introduction" at: <https://aka.ms/moc-20486d-m2-pg1>

Test Driven Development

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

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

In Microsoft Visual Studio 2017, you can define a test project, within the same solution as the main project, to store and run unit tests. After you write the tests, you can run them singly or in groups after every code change. Because MVC projects have the model, view, and controller code in separate files, it is easy to create unit tests for all aspects of application behavior.

Unified Modeling Language

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

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

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

Gathering Requirements

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

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

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

Usage Scenarios and Use Cases

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

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

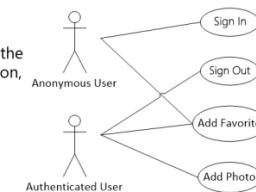
1. Roger Lengel clicks the **Add Photo** link on the main site menu.
2. Roger provides the input, **RogerL**, in the **User name** box and the password in the **Password** box to authenticate on the site.
3. Roger types the title, **Sunset**, for the photo.
4. Roger browses to the JPEG file for his new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

- Functional requirements describe how the application responds to users

- Technical requirements describe the technical features of an application, such as availability, security, or performance

- You can build functional requirements by using:

- Usage scenarios
- Use cases
- Requirements modeling in the agile methodology
- User stories in the extreme programming methodology



Sample UML Use Case Diagram

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

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

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

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

Agile Requirements Modeling

In a traditional waterfall model or iterative development model, developers and analysts investigate and record the precise and detailed functional and technical requirements at an early stage of the project. These cannot be changed later. On the other hand, in an agile development model-based project, developers recognize that requirements can change at any time during development. Requirements analysis is therefore characterized as follows:

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

User Stories in Extreme Programming

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

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

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

Planning the Database Design

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

- Logical modeling
- Physical database structure
- Working with DBAs
- Database design in agile and extreme programming

Logical Modeling

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

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

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

Physical Database Structure

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

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

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

Working with Database Administrators

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

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

Database Design in Agile Development and Extreme Programming

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

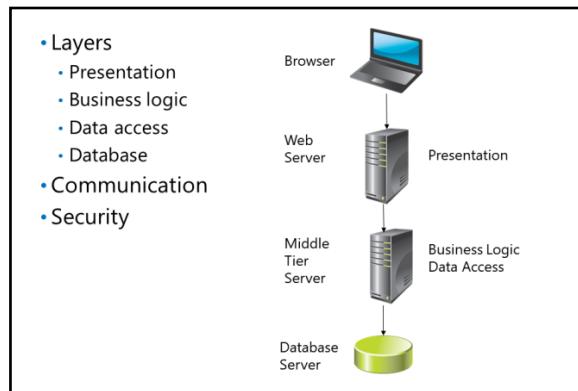
In agile development and extreme programming, the database design changes throughout the project until deployment. Therefore, developers should be able to alter the database whenever necessary without consulting DBAs or complying with complex data policies. For this reason, you should use a separate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

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

Planning for Distributed Applications

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

- Presentation layer. Components in this layer implement the user interface and presentation logic. If you are building an MVC web application, views and controllers make up your presentation layer.



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

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

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

Communication Between Layers

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

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

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

- Between the browser and presentation layer web server. In any web application, the web browser, where the presentation layer runs, communicates with the web server by using Hypertext Transfer Protocol (HTTP). If authentication is required, it is often performed by exchanging plain text credentials. You can also use Secure Sockets Layer (SSL) to encrypt this sensitive communication.
- Between the web server and the middle-tier server. The communication and security mechanisms used for communication between the web server and the middle-tier server depends on the technology that you use to build the business logic components. Two common technologies are Web API and Windows Communication Foundation (WCF).
- Between the middle-tier server and database. The middle-tier server sends T-SQL queries to the database server, which authenticates against the database by using the required credentials. These are often included in the connection string.

Planning State Management

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

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

• Client-side locations to store state data:

- Cookies
- Query strings

• Server-side locations to store state data:

- TempData
- Application state
- Session state
- Database tables

In almost all web applications, state storage is a fundamental requirement. ASP.NET provides several locations where you can store state information, and simple ways to access the state information.

However, you must plan the use of these mechanisms carefully. If you use the wrong location, you might not be able to retrieve a value when it is required. Furthermore, poor planning of state management frequently results in poor performance.

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

Client-Side State Storage

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

- Cookies. Cookies are small text files that you can pass to the browser to store information. A cookie can be stored:
 - In the memory of the client computer, in which case, it preserves information only for a single user session.
 - On the hard disk drive of the client computer, in which case, it preserves information across multiple sessions.
- Most browsers can store cookies only up to 4,096 bytes and permit only 20 cookies per website. Therefore, cookies can be used only for small quantities of data. Also, some users might disable cookies for privacy purposes, so you should not rely on cookies for critical functions.
- Query strings. A query string is the part of the URL after the question mark and is often used to communicate form values and other data to the server. You can use the query string to preserve a small amount of data from one page request to another. All browsers support query strings, but some impose a limit of 2,083 characters on the URL length. You should not place any sensitive information

in query strings because it is visible to the user, anyone observing the session, or anyone monitoring web traffic.

Server-Side State Storage

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

The following locations store state information in server memory:

- TempData. This is a state storage location that you can use in MVC applications to store values between one request and another. This information is preserved for a single request only and is designed to help maintain data across a webpage redirect. For example, you can use it to pass an error message to an error page.
- Application state. This is a state storage location that you can use to store values for the lifetime of the application. The values stored in application state are shared among all users. Application state is not an appropriate place to store user-specific values, such as preferences, because if you store a preference in application state, all users share the same preference, instead of having their own unique value.
- Session state. This is a state storage location that you can use to store information for the lifetime of a single browser session and values stored here are specific to a single user session; they cannot be accessed by other users. Session state is available for both authenticated users and anonymous users. By default, session state uses cookies to identify users, but you can configure ASP.NET to store session state without using cookies.
- Database tables. If your site uses an underlying database, like most sites do, you can store state information in its tables. This is a good place to store large volumes of state data that cannot be placed in server memory or on the client computer. For example, if you want to store a large volume of session-specific state information, you can store a simple ID value in a session state and use it to query and update a record in the database.

Planning Globalization and Localization

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

- You can use the internationally-recognized set of language codes available in browsers to present content customized to suit a user's language or region
- You can use resource files to provide a localized response suitable to a user's culture
- You can use separate views to suit each language code

Managing Browsers for Languages and Regions

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

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

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

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

Using Resource Files

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

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

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

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

Using Separate Views

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

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

Planning Accessible Web Applications

The Internet is for everyone, regardless of disabilities. Furthermore, if users with disabilities cannot easily browse your website, they might visit your competitors' websites and your company might lose business. You should therefore ensure that people with disabilities—such as those who have low vision or are hard-of-hearing—can use your website, and that their web browsers can parse the HTML content that your site presents. When you write websites, keep the following challenges and best practices in mind.

You can ensure that your content is accessible to the broadest range of users by adhering to the following guidelines:

- Provide **alt** attributes for visual and auditory content
- Do not rely on color to highlight content
- Separate content from structure and presentation code:
 - Only use tables to present tabular content
 - Avoid nested tables
 - Use **<div>** elements and positional style sheets to lay out elements on the page
 - Avoid using images that include important text
 - Put all important text in HTML elements or **alt** attributes

Users have different requirements depending on their abilities and disabilities. For example, consider the following factors:

- Users with low vision might use a standard browser, but they might increase text size with a screen magnifier so that they can read the content.
- Profoundly blind users might use a browser with text-to-speech software or text-to-Braille hardware.
- Color-blind users might have difficulty if color is used to highlight text.
- Deaf users might not be able to access audio content.
- Users with limited dexterity might find it difficult to click small targets.
- Users with epilepsy might have seizures if presented with flashing content.

You can ensure that your content is accessible to the broadest range of users by adhering to the following guidelines:

- Do not rely on color differences to highlight text or other content. For example, links should be underlined or formatted in bold font to emphasize them to color-blind users.
- Always provide equivalent alternative content for visual and auditory content. For example, always complete the **alt** attribute for images. Use this attribute to describe the image so that text-to-speech software or text-to-Braille hardware can render meaningful words to the user.
- Use markup and style sheets to separate content from structure and presentation code. This helps text interpretation software to render content to users without being confused by structural and presentation code. For example, you should apply the following best practices to display content on your webpage:
 - Avoid using tables to display the content. You should use tables only to present tabulated content. Tables can be used to render graphics and branding on a webpage, but in an accessible site, use positional style sheets to display content.
 - Avoid using nested tables. In a nested table, a table cell contains another table. These are particularly confusing to text readers because they read each table cell in a sequential order. The user is likely to become disoriented and unable to determine which cell is being read and what it means.
 - Avoid using images that include important text. Text readers cannot render text from within an image file. Instead, use markup to render this text.

 **Additional Reading:** The World Wide Web Consortium (W3C) has a project called the Web Accessibility Initiative (WAI) that promotes accessible web content. This project has published the Web Content Accessibility Guidelines. These guidelines are accepted by the web publishing industry as definitive. To read the full guidelines, go to: <https://aka.ms/moc-20486d-m2-pg5>

 **Note:** In ASP.NET Core MVC, the developer has complete control over the HTML that the web server sends to the web browser. However, you must understand accessibility principles to write accessible MVC web applications. Ensure that your entire development team is familiar with the requirements of accessibility and the related best practices.

Lesson 2

Designing Models, Controllers and Views

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

Lesson Objectives

After completing this lesson, you will be able to:

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

Designing Models

A fundamental activity in the MVC design process is designing a model. Each model class within the model represents a kind of object that your application manages.

Identifying Model Classes and Properties

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

- Model classes and properties
- Using diagrams
- Relationships and aggregates
- Entity framework
- Design in agile and extreme programming

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

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

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

- Photo. The **Photo** model class must include the **Title** and **Description** properties.

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

Using Diagrams

You can use diagrams to analyze the information your website manages and suggest a physical data model or database design for the site. You can use these diagrams to plan model classes. Each object in your diagram should be implemented as an MVC model class. The diagram can include not only the names of those classes, but also data types.

Relationships and Aggregates

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

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

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

Entity Framework

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

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



Note: Entity Framework will be covered in Module 7, “Using Entity Framework Core in ASP.NET Core”.

Design in Agile and Extreme Programming

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

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

Designing Controllers

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

Identify Controllers and Actions

In an ASP.NET Core MVC web application, there is usually one controller for each model class.

Conventionally, if the model class is called "Photo", the controller is called "PhotoController". If you follow this convention in your design, you can use the MVC default routing behavior to select the right controller for a request.

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

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

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

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

- Identify controllers and actions
- Design in agile and extreme programming

Controller	Action	Description
Photo	AddPhoto (GET)	The AddPhoto action for GET requests creates a new instance of the Photo model class, sets default values such as the created date, and passes it to the correct view.
	AddPhoto (POST)	The AddPhoto action for POST requests calls the Photo model class methods to save the photo values to the database and redirects the browser to the DisplayGallery action.
	DisplayGallery (GET)	The DisplayGallery action for GET requests displays all the photos stored in the database.

Controller	Action	Description
User	Logon (GET)	The Logon action for GET requests displays a view into which an anonymous user can enter credentials.
	Logon (POST)	The Logon action for POST requests checks user credentials against the membership database. If the credentials are correct, the Logon action authenticates and redirects the user to the originally requested page.

Design in Agile and Extreme Programming

Similar to the design of models, you will only make generalized plans for controllers during the design phase of an agile development or extreme programming project. You should specify only a few actions at this stage.

Designing Views

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

In an ASP.NET Core MVC web application, the user interface is created by building views. There is a many-to-one relationship between MVC controllers and views. For example, a controller might use one view to display a single photo, another view to display several photos, and a third view to enable users to upload new photos. Each view corresponds to a webpage that the application can display to the user, although it can display different data. For example, the **PhotoDetails** view can display different photos, depending on the ID parameter it receives.

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

- Views
- Layouts
- Partial views and view components
- Design in agile and extreme programming



 **Note:** Layouts will be covered in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

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

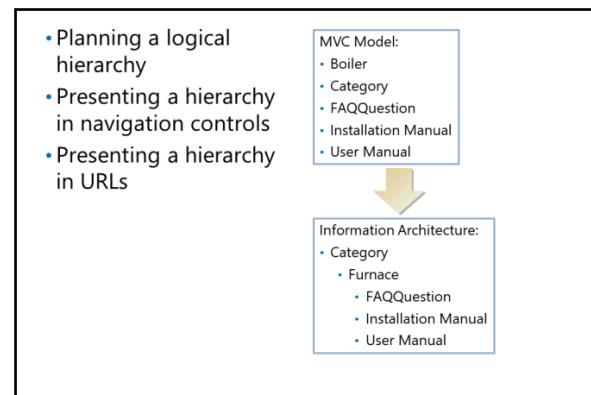
 **Note:** Partial views and view components will be covered in Module 5, "Developing Views".

Design in Agile and Extreme Programming

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

Information Architecture

When the information your web application manages is complex and multi-faceted, it is easy to present objects in a confusing way. Unless you think carefully about the way users expect information to be structured and how they expect to navigate to useful content, you might unintentionally create an unusable web application. During development, when you work with a limited amount of data, this confusion might not become apparent. Then, when real-world data is added to your database at the time of deploying to production, it becomes clear that the web application is confusing. You can avoid this eventuality by planning the information architecture.



Information architecture is a logical structure for the objects your web application manages. You should design such architecture in a way that users can find content quickly without understanding any technical aspects of your web application. For example, users should be able to locate a technical answer about their product without understanding the database structure or the classes that constitute the model.

Example Scenario: A Technical Documentation Website

To understand the need for information architecture, consider a web application that presents technical information about a company's products for customers and engineers. In this example, the company manufactures domestic heating furnaces, which are installed in homes by qualified heating engineers. The web application is intended to enable customers to locate instructions, hints, and tips. This web application is also intended to enable engineers to obtain the technical documentation on installing and servicing furnaces.

The development team has identified the following simple user stories:

- Customers have a certain problem with their boilers. They want to find a specific answer in the FAQ that solves the problem. They know the boiler product name and the fuel, but not the product number. They visit the web application and navigate to the boiler name. They click the **FAQ** link for the boiler and locate the answer.
- Engineers need the latest installation manual for a boiler. They know the boiler product number, product name, and fuel type. They visit the site and navigate to the boiler name. They click the **Manuals** link and locate the installation manual.
- Engineers have web applications, which they want to link to a specific boiler name. You want to ensure that the URL is simple and readable for customers and engineers.

You have already planned the following model classes:

- Furnace. Each furnace object is a product manufactured and sold by the company.

- Category. Categories organize the furnaces by type. For example, you can have categories such as oil-fired, gas-fired, and solid fuel.
- FAQ. Each FAQ relates to a single furnace. Each furnace can have many questions. The class includes both the **Question** and the **Answer** properties.
- Installation Manual. Each furnace has a single installation manual in the form of a PDF document.
- User Manual. Each furnace has a single user manual in the form of a PDF document.

Planning a Logical Hierarchy

You can see from the user stories that FAQ and manuals are both accessed by navigating to the relevant product. You can also see that the company has different products, and both customers and engineers know the fuel type for a particular furnace. Therefore, you can organize furnaces in categories by fuel type. The following list shows a logical hierarchy of objects, which helps both the customers and engineers find the information they need by clicking through each level:

- Category
 - Furnace
 - FAQ
 - User Manual
 - Installation Manual

Presenting a Hierarchy in Navigation Controls

The information architecture you design should be presented on webpages in the form of navigation controls. Common approaches to such controls include:

- Site Menus. Most websites have a main menu that presents the main areas of content. For simple web applications, the main menu might include a small number of static links. For larger web applications, when users click a site menu link, a submenu appears.
- Tree Views. A tree view is a menu that shows several levels of information hierarchy. Usually, users can expand or collapse objects at each level, to locate the content they require. Tree views are useful for presenting complex hierarchies in navigable structures.
- Breadcrumb Trails. A breadcrumb trail is a navigation control that shows the user where they are in the web application. Usually a breadcrumb trail shows the current pages and all the parent pages in the hierarchy, with the home page as the top level page. Breadcrumb trails enable you to understand how a page fits in with the information architecture shown in menus and tree views.

The types of navigation controls you build in your web application depend on how you expect users to find information.

Presenting a Hierarchy in URLs

You can increase the usability of your web application by reflecting the information architecture in the URLs, which the users see in the address bar of the web browser. In many web applications, URLs often include long and inscrutable information such as Globally Unique Identifiers (GUIDs) and long query strings with many parameters. Such URLs prevent users from manually formulating the address to an item, and these URLs are difficult to link to a page on your web application. Instead, URLs should be plain and comprehensible, to help users browse through your content.

In MVC web applications, the default configuration is simple, but it is based on controllers, actions, and parameters. The following are some example URLs that follow this default pattern:

- <http://site/Furnace/Details/23>: This URL links to the **Furnace** controller and the **Details** action, and it displays the furnace with the ID 23.

- <http://site/FAQQuestion/Details/234>: This URL links to the **FAQQuestion** controller and the **Details** action, and it displays the FAQ with the ID 234.
- <http://site/InstallationManual/Details/3654>: This URL links to the **InstallationManual** controller and the **Details** action, and it displays the manual with the ID 3654.

As you can see, the web application user is required to understand how controllers, actions, and action parameters work, to formulate URLs themselves. Instead, users can use URLs that are easier to understand, such as the following, because they reflect the information hierarchy:

- <http://site/OilFired/HotBurner2000>: This URL links to a furnace by specifying the fuel category and the product name. Customers and engineers can understand these values.
- <http://site/OilFired/HotBurner2000/HowToRestartMyFurnace>: This URL links to an FAQ by specifying the furnace name to which the question relates.
- <http://site/OilFired/HotBurner2000/>: This URL links to the Installation Manual by specifying the furnace name to which the manual relates.

As you can see, these URLs are easy for customers and engineers to understand, because the URLs are based on a logical hierarchy and the information that the users already have. You can control the URLs that your ASP.NET web application uses, by configuring the ASP.NET routing engine.



Note: Routing will be covered in Module 4, "Developing Controllers".

Lab: Designing ASP.NET Core MVC Web Applications

Scenario

Your team has chosen ASP.NET Core MVC 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 Core MVC web application that meets a set of functional requirements.
- Record the design in an accurate, precise, and informative manner.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD02_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD02_LAK.md.

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.

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.

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. Define the view.
2. Design the single photo view.
3. Design the gallery view.

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.

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.

Review Question

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 well 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 during the development phase. Project creep results in projects that are over-budget and late.

Tools

You can use Microsoft Office Visio and Visual Studio 2017 to create design diagrams.

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.

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.	Use agile development and extreme programming methodologies. Such methodologies, which are based on the real-world assumption that requirements will change frequently, are proven to prevent the time wastage that often occurs when complete designs are altered during the development phase.

Module 3

Configure Middleware and Services in ASP.NET Core

Contents:

Module Overview	03-1
Lesson 1: Configuring Middleware	03-2
Lesson 2: Configuring Services	03-14
Lab: Configuring Middleware and Services in ASP.NET Core	03-28
Module Review and Takeaways	03-30

Module Overview

ASP.NET Core is a framework that allows us to build many different kinds of applications. In this module, you will learn how to leverage the ASP.NET Core framework to handle requests and responses via existing, and custom middleware, and how to configure services for use in middleware and throughout other parts of the application, such as controllers.

A middleware is a segment of code that can be used as part of the request and response pipeline that allows us to handle them according to any relevant parameter. This potentially allows multiple separate requests to be handled in a completely different fashion and receive separate responses.

Services are classes that expose functionality which you can later use throughout different parts of the application, without having to keep track of scope manually in each individual location and instantiate any dependencies. This is done by using Dependency Injection.

Dependency Injection is a technique used by ASP.NET Core that allows us to add dependencies into the code without having to worry about instantiating objects, keeping them in memory, or passing along required dependencies. This allows the application to become more flexible and to reduce potential points of failure whenever you change a service.

Objectives

After completing this module, you will be able to:

- Use existing middleware to set up an ASP.NET Core application.
- Create your own middleware and use it to define custom behavior.
- Understand the basic principles behind Dependency Injection, and how it is used in ASP.NET Core.
- Know how to create a custom service, configure its scope, and inject it to both middleware and ASP.NET Core MVC controllers.

Lesson 1

Configuring Middleware

Middleware is code, which can be attached to the request and response pipeline and is used to perform manipulations on all the requests and responses occurring throughout the server. Without any middleware, you are unable to actually handle any client requests, and you will simply get an error page.

By using middleware, you can inspect the request, and make important decisions on how to handle the current step. Middleware can be used to perform any operation that you can possibly see as relevant to the current request, including writing to the response. However, not all middleware will directly influence the response. At any time, that should you decide it is required, you can short-circuit the middleware pipeline and return a finalized response.

In this lesson, you will learn about the importance of the **Startup** class in configuring the middleware that you wish to use, cover the basics behind the concept of the middleware, learn how to create custom middleware, and how to implement them. Finally, you will be introduced to the **UseStaticFiles** middleware, as an example for built-in middleware, which allows us to serve static files such as HTML, CSS and JavaScript files to the end users.

It is important to understand that the flow of the middleware and their order determines how the application is run, and that understanding how middleware function can help us build more consistent behavior for the application while avoiding undesirable edge cases.

Lesson Objectives

After completing this lesson, you will be able to:

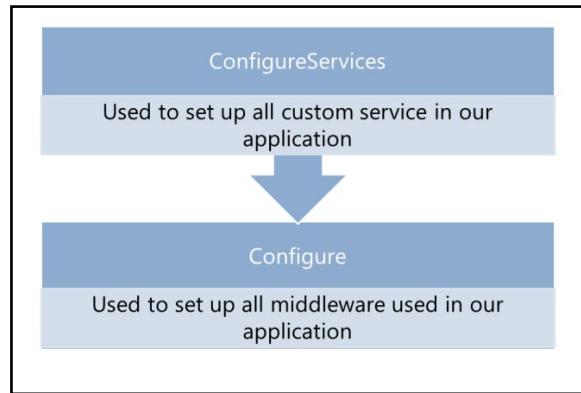
- Describe the basic purpose of the **Startup** class, in relation to middleware.
- Describe the usage of middleware, and the correct order to manage their pipeline.
- Use existing middleware to extend an ASP.NET Core web application's capabilities.
- Configure your own custom middleware, thereby extending the capabilities of ASP.NET Core to suit your requirements.
- Serve static files to users of the application.

Application Startup

At the basis of every ASP.NET Core application is the **Startup** class. This class is created by default in every new project and is used to help us configure the application. It is used for configuring middleware, which is performed in the **Configure** method, and for configuring services, which is performed in the **ConfigureServices** method.

The **Startup** class is called and run from within the **program.cs** file.

As part of startup, the **ConfigureServices** method will be called first if it is present, before calling the **Configure** method, which may have dependencies on the services that were declared in **ConfigureServices**.



 **Best Practice:** By default, the startup class will use the name **Startup**. While this can be changed, and you can use any name you wish, it is a best practice to use the name **Startup**. This allows people unfamiliar with the application to find it easily and to add required middleware and services.

The following code shows a **Startup** class with a middleware:

The Startup class

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

In this example, the string **Hello World!** is always displayed in the browser, no matter which relative URL the user requests.

The ConfigureServices Method

The **ConfigureServices** method is an optional method where services that will be injected during the application lifespan can be registered. This will be covered in greater detail in Lesson 2, "Configuring Services". It can also be used to set up various additional configuration options for the application.

The **ConfigureServices** method receives a parameter that implements the **IServiceCollection** interface. This parameter is used to add services to the application.

The following code shows a custom service being added to the application service collection:

A ConfigureServices example

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IService, Service>();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

 **Note:** Note that some services are predefined in the framework and will not need to be declared. For example, **ILogger**, and **IHostingEnvironment**, both of which will be covered in Module 10, "Testing and Troubleshooting".

The Configure Method

The **Configure** method is where the middleware pipeline can be defined. It is possible to add both existing middleware, as well as custom ones. The **Configure** method supports Dependency Injection, allowing us to inject any services that have been set up and use them inside the middleware logic. In general, the **Configure** method is where the order of handling requests and responses is determined. To use the **Configure** method, you need to pass to it an **IApplicationBuilder** parameter. This parameter is used for adding middleware to the pipeline.

The following code shows a **Configure** method where a middleware is added. The middleware calls the service that is registered in the **ConfigureServices** method:

A Configure example

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IService, Service>();
    }

    public void Configure(IApplicationBuilder app, IService service)
    {
        app.Use(async (context, next) =>
        {
            service.DoSomething();
            await next.Invoke();
        });

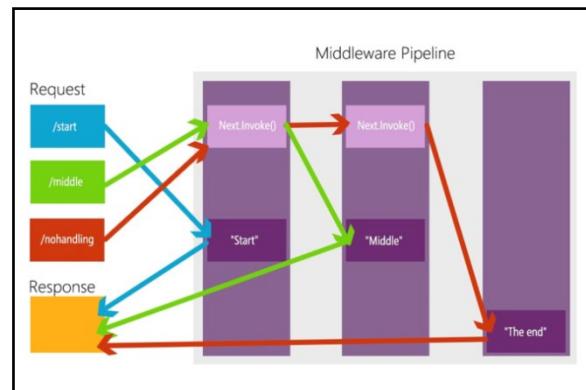
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

Middleware Fundamentals

Middleware allows us to add code to manipulate the request and response pipeline in ASP.NET Core applications. Every middleware in the pipeline can interact with the provided request, and write to the response, creating the desired result.

 **Note:** Middleware can be both custom-made, and premade middleware provided by the ASP.NET Core framework. For example, ASP.NET Core MVC is made available by using the **UseMvc** or **UseMvcWithDefaultRoute**

middleware. These middleware will be covered in further detail in Module 4, "Developing Controllers".



The following code shows a basic middleware pipeline:

A middleware example

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

In this example, the string **Hello World!** will be displayed in the browser on every request, no matter which relative URL the user requests.

Middleware is intended to work in a pipeline with each item inspecting the request, performing an action, or a manipulation on the response object and on completing its run, either calling the next middleware in the pipeline or if appropriate for the scenario, short-circuiting the pipeline and sending the finalized response. Short-circuiting should be done whenever the handler deems the request ended.

 **Note:** Scenarios where you may want to short circuit the request:

- When you match the correct route for the request.
- When an error has been discovered in the request.
- When you want to prevent the default handler.

Every single instance of middleware has access to an **HttpContext** parameter from the **Microsoft.AspNetCore.Http** namespace. This parameter allows us to access and manipulate information regarding the current request and response, and to many other details such as the authentication details, the current user, and more. At this point of the course, the main focus will be on the **Request** and **Response** properties.

The **Request** property allows us to investigate details about the request made to the server, including the **Query** property, which allows us to read the **Path** and **Body** properties from the query string as a collection of key-value pairs. The **Path** property allows us to find the relative path within the application. The **Body** property is data sent as part of the request body and more. For this lesson, the **Path** and **Query** properties will mainly be used.

The **Response** property allows us to manipulate the response which will be sent back to the client. It allows us to write to it using the **WriteAsync** method, or alternatively allows us to directly manipulate the response **Body**, **ContentType**, or even manually set **StatusCodes**. Setting the **StatusCodes** allows us to return custom errors. In this lesson, you will mainly be using **WriteAsync** in order to communicate with the client.

The following code demonstrates middleware that examines the query string:

Context response query string

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        if (context.Request.Query.ContainsKey("id"))
        {
            await context.Response.WriteAsync($"The ID in the Query string is: {context.Request.Query["id"]}, ");
        }
        await context.Response.WriteAsync($"The path is: {context.Request.Path.Value}");
    });
}
```

```
}
```

The following table shows the requests and responses for the following relative URL paths using the previous code.

Request	Response
{baseUrl}	The path is: /
{baseUrl}/Example/Path	The path is: /Example/Path
{baseUrl}?id=1	The ID in the Query string is: 1 The Path is: /
{baseUrl}/Example/Path?id=1	The ID in the Query string is: 1 The Path is: /Example/Path

Middleware always run in the order they are defined. Additionally, all middleware operate in sequence, and are of several potential types:

- **Use.** Middleware added with **Use**, support a parameter named **next**, which is a reference to the next middleware in the pipeline. You can short-circuit the pipeline by not invoking **next**, although all **Use** middleware should preferably support at least one flow in which they call **next.Invoke()** to proceed to the next middleware in the pipeline.
- **Run.** Unlike **Use**, the **Run** middleware will always be the final middleware in the pipeline. It does not support the **next** parameter, and any middleware appearing after it will never be called.
- **Map.** A third variation is the **Map** middleware. It does not continue with the current pipeline, instead, if the relative path is matched to the **Map** middleware, it will continue down a new pipeline, which is provided as a delegate to the **Map** middleware. Note that since **Map** middleware creates its own pipeline it is not affected by the **Run** middleware which occurs after it and can support a call to **Run** at the end of its own pipeline.

The following code demonstrates a **Configure** method with only a **Run** middleware:

A Run middleware example

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Inside run middleware");
    });
}
```

In this example, the **Inside run middleware** string will be displayed in the browser on every request, no matter which relative URL the user requests.

The following code demonstrates the **Use** middleware:

A Use middleware example

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Inside use middleware => ");
        await next.Invoke();
    });

    app.Run(async (context) =>
    {
```

```

        await context.Response.WriteAsync("Inside run middleware");
    });
}

```

In this example the **Inside use middleware => Inside run middleware** string will be displayed in the browser on every request, no matter which relative URL the user requests.

The following example demonstrates a **Use** middleware that can short-circuit the middleware pipeline:

A Use middleware with a short circuit

```

public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        if (context.Request.Query.ContainsKey("shortcircuit"))
        {
            await context.Response.WriteAsync("Inside use middleware");
        }
        else
        {
            await next.Invoke();
        }
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Inside run middleware");
    });
}

```

If a user requests a URL with a query string parameter, which has the name **shortcircuit** (value isn't required) the **Inside use middleware** string will be displayed in the browser. On any other request, the **Inside run middleware** string will be displayed in the browser.

The following code demonstrates a **Map** middleware:

A Map middleware example.

```

public void Configure(IApplicationBuilder app)
{
    app.Map("/Map", (map) =>
    {
        map.Run(async (context) =>
        {
            await context.Response.WriteAsync("Run middleware inside of map middleware");
        });
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Inside run middleware");
    });
}

```

If a user requests a URL with a relative path starting with **/Map**, a new pipeline will be entered in which the **Run middleware inside of map middleware** string will be displayed in the browser. If the relative path does not begin with **/Map**, the **Inside run middleware** string will be displayed in the browser.

 **Best Practice:** It is a good idea to always end the **Configure** method with a **Run** middleware. This allows us to always ensure that the client receives a response from the server, even if it is a generic response. Otherwise, the user may encounter a generic error page.

Order of Middleware

It is important to remember that middleware runs in the order in which they were added to the pipeline, therefore it is important to keep the following in mind:

- A **Run** middleware should always be present at the very end of the pipeline. All middleware defined after the **Run** middleware will never be run.
- Every application should only have a single **Run** middleware inside a specific pipeline. Remember that by using **Map** you create a new pipeline, which means the **Run** middleware inside of a **Map** pipeline is separate from the main pipeline.
- Whenever multiple middleware share the same condition, it is important to order them to handle the pipeline in the desired way and be mindful of the possibility of the pipeline being short-circuited.

The following code demonstrates three middleware operating in a linear order:

A middleware order example

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("First middleware => ");
        await next.Invoke();
    });

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Second middleware => ");
        await next.Invoke();
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Final middleware.");
    });
}
```

In this example the **First middleware => Second middleware => Final middleware** string will be displayed in the browser on every request, no matter which relative URL the user requests.

The following code demonstrates the detrimental effect of the **Run** middleware running before other middleware:

An example of Run middleware short circuiting all other middleware

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Final middleware.");
    });

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("First middleware. =>");
        await next.Invoke();
    });

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Second middleware. =>");
        await next.Invoke();
    });
}
```

```
}
```

In this example, only the **Final middleware** string will be displayed in the browser on every request, no matter which relative URL the user requests.

The following code shows an example of the **Use** middleware short-circuiting the pipeline:

Use middleware short circuiting

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("First middleware => ");
        await next.Invoke();
    });

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Second middleware => ");
        if (!context.Request.Query.ContainsKey("shortcircuit"))
        {
            await next.Invoke();
        }
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Final middleware.");
    });
}
```

If a user requests a URL with a query string parameter with the name **shortcircuit**, the **First middleware => Second middleware => string** will be displayed in the browser. If the parameter is not present in the request, the **First middleware => Second middleware => Final middleware** string will be displayed in the browser.

Demonstration: How to Create Custom Middleware

In this demonstration, you will learn how to add middleware to the pipeline.

Demonstration Steps

You will find the steps in the section "Demonstration: How to Create Custom Middleware" on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD03_DEMO.md#demonstration-how-to-create-custom-middleware.

Working with Static Files

An important part of every web application is the ability to serve static files, whether they are images to be displayed on the page, HTML pages for the users to peruse, JavaScript files to customize interactions with the user, or CSS stylesheets to better present the application.

Default Static File Serving

To serve static files within an ASP.NET Core application, a specific preexisting middleware will need to be used. The middleware required is **UseStaticFiles**. By calling **UseStaticFiles**, the application will automatically match relative paths to files inside the **wwwroot** folder of the application and serve them as the response for the current request. Any file placed outside the **wwwroot** folder, will not be served and will be ignored by the server.

- Static files are files that do not change at run time
- In ASP.NET Core applications static files can be served to clients by using the **UseStaticFiles** middleware

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

 **Best Practice:** It is important to note, that what you learned about the order of middleware in Topic 2, "Middleware Fundamentals", is still relevant when working with **UseStaticFiles**. It is generally considered a best practice to call **UseStaticFiles** early in the middleware pipeline. However, you must take care that the paths served in **UseStaticFiles** do not overlap with paths you expect to handle in other middleware since this can result in unexpected behavior.

The following code is an example of calling the **UseStaticFiles** middleware:

Calling **UseStaticFiles** middleware

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Not a static file.");
    });
}
```

If the user requests a URL with a relative path that matches the directory structure under **wwwroot**, the specified file will be sent back as the response. Otherwise, the **Not a static file.** string will be displayed in the browser.

Serving Static Files from Outside the **wwwroot**

In some cases, you may want to serve static files from a different folder, rather than **wwwroot**. To do this, you will need to call the **UseStaticFiles** middleware with a **StaticFileOptions** parameter. By setting the **FileProvider** property, you can assign **PhysicalFileProvider**, which allows you to choose a file path. All files under this path will then be served as if they were on **wwwroot**. It is important to note, that this replaces **wwwroot**. If you wish to host static files on both, you will need to call **UseStaticFiles** twice. Once without parameters, and the second time with the **StaticFileOptions** parameter.

 **Best Practice:** It is a best practice to keep **wwwroot** as the folder from which static files will be will hosted. However, should the need arise to host static files for a third-party component, the option exists.

The following code is an example of hosting static files from **wwwroot** and from a folder named **StaticFolder**:

Hosting static files from multiple locations

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new
PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), "StaticFolder"))
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Not a static file.");
    });
}
```

If a user requests a URL with a relative path that matches with a path under **wwwroot** or **StaticFolder**, the specified file will be sent back as the response. Otherwise, the **Not a static file.** string will be displayed in the browser.

In this example, if an attempt is made to access any files under either **wwwroot** or **StaticFolder**, they will both be accessible. An important note is that if the same file path exists in both, the one in **wwwroot** takes precedence as the **UseStaticFiles** middleware that configures **wwwroot** is called first.

 **Note:** **Path.Combine** is a method that is designed to combine multiple file system paths into a single valid path string.
Directory.GetCurrentDirectory is used to retrieve the current working directory of the application at run time.

Serving Files under a Separate Relative Path

Sometimes you may wish to group the static files under a separate relative path. This can be particularly useful to help avoid conflicts. By doing this, you can ensure that a specific relative path is kept separate from the remaining middleware pathing logic. This can also be done by supplying static files with **StaticFileOptions** while supplying the **RequestPath** property. The request path must always begin with a / character and can contain several hierarchies if you so wish. After doing this, to use the static files, the application relative path matches the request path and then it will check for a match in the remaining path later.

The following code demonstrates serving static files on separate relative paths:

Static files on separate relative paths

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions()
    {
        RequestPath = "/wwwroot"
    });

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new
PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), "StaticFolder")),
        RequestPath = "/StaticFiles/MyFolder"
    });
}
```

```

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Not a static file.");
    });
}

```

The following table shows the requests and responses for the following relative URL paths using the previous code:

Request URL	File exists	Response
{baseUrl}/page.html	Irrelevant	File will not be served.
{baseUrl}/StaticFiles/MyFolder/page.html	Yes	page.html from StaticFolder
{baseUrl}/StaticFiles/MyFolder/page.html	No	File will not be served.
{baseUrl}/wwwroot/page.html	Yes	page.html from wwwroot
{baseUrl}/wwwroot/page.html	No	File will not be served.
{baseUrl}/StaticFiles/MyFolder/views/page.html	Yes	page.html from StaticFolder/views
{baseUrl}/StaticFiles/MyFolder/views/page.html	No	File will not be served.
{baseUrl}/wwwroot/views/page.html	Yes	page.html from wwwroot/views
{baseUrl}/wwwroot/views/page.html	No	File will not be served.
{baseUrl}/other/page.html	Irrelevant	File will not be served.



Note: Note that this will allow us to potentially have the same paths under different directories which are served without conflicts.

Problems with Serving Static Files

While static files are a very convenient means of getting files to the user, there are some drawbacks to using **UseStaticFiles**.

- All files served by using **UseStaticFiles** will always be available, there is no way to require authorization for them, and they will be available to all users.
- Files served from **UseStaticFiles** will potentially be able to reveal information about the application structure, as all of them and their paths are accessible.
- Depending on the hosting operating system, these files may or may not be case sensitive.

Therefore, it is important that any file you wish to protect should not be inside folders you are serving. In particular, any cs code, or cshtml pages (cshtml pages will be covered in Module 5, "Developing Views") should never be placed under a folder designated for serving to the client.

Demonstration: How to Work with Static Files

In this demonstration, you will learn how to use the **UseStaticFiles** middleware and its behavior under certain conditions, such as, serving static files and handling files which are not found.

Demonstration Steps

You will find the steps in the section "Demonstration: How to Work with Static Files" on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD03_DEMO.md#demonstration-how-to-work-with-static-files.

Lesson 2

Configuring Services

Services are classes that can be reused easily in multiple locations without having to worry about instantiating their various dependencies and sub-dependencies. By doing this, it is possible to greatly decrease the overhead involved in developing code by reducing management and potential points of failure. This is facilitated via a technique known as Dependency Injection. As a result, large segments of code can easily be reused in different controllers, and even inside the **Configure** method of the **Startup** class.

In this lesson, you will learn about how to create and use services by leveraging the strengths of Dependency Injection to create versatile code, which does not require frequent upkeep. You will begin by delving into Dependency Injection, what it means and how it works, and then go into a demonstration on how to put it into effect, allowing the addition of custom services into the **Configure** method, and how to use them in other services that will be created. You will then take a look at ASP.NET Core MVC controllers (Controllers will be covered in Module 4, “Developing Controllers”), and learn how by using middleware and services it is possible to run an ASP.NET Core MVC application, and how you can reuse services inside controllers. Finally, you will then also learn how to best manage the service lifetime and maintain a logical lifetime for the various services used by the application.

It is important to understand the importance of how Dependency Injection can help to create a cleaner, more easily maintainable code, while providing us with relevant tools for managing them, and making the overall code easier to test and to learn how you can use the services at any point in the application.

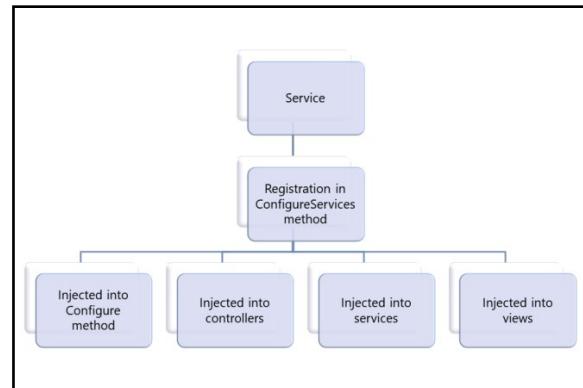
Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basics behind Dependency Injection and explain its purpose.
- Create new services and add them to the application.
- Configure the correct lifetime scope for each service as required.
- Be able to inject services to controllers, other services, or the **Configure** method, and use it from within the middleware.

Introduction to Dependency Injection

Dependency Injection is a technique by which it is possible to facilitate separation of concerns in the application. Normally, to utilize a class in the application, you will need to be able to also manage any classes it depends on. This can cause a situation where whenever you instantiate a class, you may need to pass along to it a large number of parameters, which are then used to instantiate the class and potentially instantiate additional classes it depends on.



The following code is an example of classes with a dependency chain:

Dependency chain

```
public class MyClass
{
    private MyDependency _myDependency;

    public MyClass(MyDependency myDependency)
    {
        _myDependency = myDependency;
    }
}

public class MyDependency
{
    private MySubDependency _mySubDependency;
    public MyDependency(MySubDependency mySubDependency)
    {
        _mySubDependency = mySubDependency;
    }
}

public class MySubDependency
{
    public MySubDependency()
    {

    }
}
```

The following code is an example of trying to instantiate MyClass:

Instantiating MyClass

```
public class SomeClass
{
    private MyClass _myClass;

    public SomeClass()
    {
        MySubDependency subDependency = new MySubDependency();
        MyDependency dependency = new MyDependency(subDependency);
        _myClass = new MyClass(dependency);
    }
}
```

As you can see, normally when you try to instantiate this chain of dependencies, you need to also take care of its sub-dependencies and if the class you have defined is required in other places, you will need to instantiate it in there as well.

By implementing Dependency Injection, you are able to simplify the process and adopt a more loosely coupled approach. By explicitly defining interfaces you wish to work within the constructor, you are able to specify only the requirements for the current class and you don't need to worry about any possible sub-dependencies or making any changes should any sub dependency code change.

The following code is a more loosely coupled variant of the dependency chain seen previously:

Loosely coupled dependency chain

```
public interface IMyClass
{
}

public interface IMyDependency
{
}

public interface IMySubDependency
{
}

public class MyClass : IMyClass
{
    private IMyDependency _myDependency;

    public MyClass(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }
}

public class MyDependency : IMyDependency
{
    private IMySubDependency _mySubDependency;
    public MyDependency(IMySubDependency mySubDependency)
    {
        _mySubDependency = mySubDependency;
    }
}

public class MySubDependency : IMySubDependency
{
    public MySubDependency()
    {
    }
}
```

The following is an example of managing code with Dependency Injection:

A dependency injection example

```
public class SomeClass
{
    private IMyClass _myClass;

    public SomeClass(IMyClass myClass)
    {
        _myClass = myClass;
    }
}
```

This code demonstrates how the previous code segment could function if Dependency Injection was used. As you can see rather than needing to instantiate the various sub-dependencies, both **SomeClass** and **MyClass** only require their own direct dependencies, creating a far more loosely coupled relationship. Additionally, if you wanted to give any of these classes additional dependencies, it will not affect anything else.

By using interfaces, you are able to declare the behavior that you desire to implement, rather than a specific implementation. By using a Dependency Injection container, you can allow the application to support Dependency Injection, freeing us from having to manage multiple components whenever a change occurs and allowing us to work on a more abstract level.

The Dependency Injection container is a class that lies at the heart of the mechanism. This class is a factory class with the sole purpose of instantiating classes as they are needed and providing them to other classes as needed. This encompasses not only the **Configure** method but also extends to controllers (Controllers will be covered in Module 4, "Developing Controllers"), views (Views will be covered in Module 5, "Developing Views"), and even other services. Whenever a class in the application is instantiated by using a Dependency Injection container, it will request any of its other dependencies, allowing us to work exclusively with the class itself without having to worry about having to manage any of its dependencies.

By default, ASP.NET Core uses a simple built-in Dependency Injector container, which gives us the support of constructor injection. You can use the **IServiceProvider** interface to interact with the container, and the most prominent interaction is through the **ConfigureServices** method of the **Startup** class, which allows you to register additional services that you wish to run throughout the application.

 **Note:** One of the biggest advantages granted us by Dependency Injection is that due to the code being implemented via services, you are able to very easily test each individual component. Services, controllers, and more are all easily tested when you can easily provide mock data via the interface, rather than having to support tests covering multiple components. This ensures that testing is more thorough and more extensive while being easier to maintain.

Using the Startup Class to Configure Services

As you saw in Lesson 1, "Configure Middleware", the **Startup** class exposes both the **Configure** method, which is used to set up the middleware pipeline and the **ConfigureServices** method, which allows us to determine which services you want to let the Dependency Injection container manage. The method itself supports both built-in methods for loading preconfigured services and collections for loading multiple services such as **AddMvc**. **AddMvc** loads multiple services that are not required for ASP.NET Core, as well as injecting any number of custom services you may wish to inject.

- Any class can act as a service
- By using the **ConfigureServices** method, you can register any service you would like to use in the application
- By using Dependency Injection in the **Configure** method you can utilize the services inside the middleware

Injecting Custom Services

To inject a custom service, you will first need to create a service. This can be done easily as services in ASP.NET Core are simple classes which implement any interface chosen by the developer.

The following is an example of a basic service:

A basic service

```
public interface IMyService
{
    string DoSomething();
}

public class MyService : IMyService
```

```
{  
    public string DoSomething()  
    {  
        return "something";  
    }  
}
```

In this example, a class called **MyService** is created, which implements the **IMyService** interface. This class implements a **DoSomething** method, which when called returns the **something** string. This example will be used in the demos throughout this topic.

After creating the service, you will need to register it in the **ConfigureServices** method of the **Startup** class. This can be done by using the **services** parameter, which implements the **IServiceCollection** interface. This method exposes to us several methods that can be used to add the service to the Dependency Injection container. The main methods for adding custom services are **AddSingleton**, **AddTransient**, and **AddScoped**. You will learn about them and their various differences in the Topic 5, "Service Lifetime". But for now, you will use **AddSingleton**.

To register the service, you will call the **AddSingleton** method. You will provide the type parameters for the interface that you want to declare and the type of the class that you want to instantiate.

The following code is an example of registering a custom service:

Registering a custom service.

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddSingleton<IMyService, MyService>();  
  
    public void Configure(IApplicationBuilder app, IMyService myService)  
    {  
        app.Run(async (context) =>  
        {  
            await context.Response.WriteAsync(myService.DoSomething());  
        });  
    }  
}
```

In this example, you can see that in the **ConfigureServices** method you are registering a service with the **MyService** class, which implements the **IMyService** interface. Later, you can see that in the **Configure** method, you have injected the service and used it inside of the middleware. In this example, the **something** string will be displayed in the browser on every request, no matter which relative URL the user requests.

 **Best Practice:** It is a best practice to name interfaces with the same name as the class with an **I** at the beginning. However, it would be possible to register a service with completely different class name and interface name, as long as the class implements the interface. Doing so will create confusing code that is difficult to maintain.

Injecting System Services

Along with custom services, there are various services that are not registered by default and which you can add to the application to enhance its functionality. To run an ASP.NET Core MVC application, you will use the **AddMvc** method, which sets up various services that are required to correctly run an ASP.NET Core MVC application. Unlike custom services, **AddMvc** runs internal code on **IServiceCollection**, which loads multiple services, including services for Razor Pages and services for handling views.

The following code is an example of using the **AddMvc** method:

The AddMvc usage

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IMyService, MyService>();
    services.AddMvc();
}
```

This example demonstrates the usage of **AddMvc**, as well as, adding the service that was configured earlier.

 **Note:** If you look into **IServiceCollection**, you will see that it exposes both the **AddMvc** and **AddMvcCore** methods. It is important to note that **AddMvcCore** contains only the very basic components required to run an ASP.NET Core MVC application and lacks several additional services that are used for various additional functionalities, such as Cross Site Scripting handling and the Razor View engine. For this course, you will mainly use **AddMvc**.

Injecting Services into Other Services

Sometimes you can encounter a case in which one of the services has a dependency on another. Common reasons for this can include access to specific data, reusable methods and accessing resources that can bottleneck the application. In all of those cases, having multiple services performing the same functionality would create a hard to maintain environment, with a lot of repeated code and services with too many responsibilities. By using Dependency Injection, you can resolve this issue by injecting services into each other.

To inject a service into another, all you require is to add a constructor within the service, with a parameter type matching the interface you want to instantiate. After this is done, inside the constructor you can perform whatever logic you need from the dependency, including storing it locally inside of the class.

The following code shows a service being injected into another service:

Dependency injection between services

```
public interface IFirstService
{
    string GoFirst();
}

public interface ISecondService
{
    string GoSecond();
}

public class FirstService : IFirstService
{
    public string GoFirst()
    {
        return "Going first";
    }
}

public class SecondService : ISecondService
{
    private IFirstService _firstService;

    public SecondService(IFirstService firstService)
    {
        _firstService = firstService;
    }
}
```

```
public string GoSecond()
{
    return $"({_firstService.GoFirst()} - Going Second";
}
```

In this example, you can see that **SecondService** stores a reference to **FirstService**, and whenever **GoSecond** is called, the **Going First – Going Second** string will be returned.

The following code demonstrates registration and usage of services with injection:

Service dependency registration and usage

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IFirstService, FirstService>();
    services.AddSingleton<ISecondService, SecondService>();

public void Configure(IApplicationBuilder app, ISecondService secondService)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync(secondService.GoSecond());
    });
}
```

In this example, you can see registration for both services from the previous example, alongside injecting **SecondService** into the **Configure** method and using it inside the middleware. For all requests, **Going First – Going Second** will be displayed in the browser.

 **Best Practice:** Services inside **ConfigureServices** can be registered in any order, unlike middleware, and are resolved by the Dependency Injection container. However, it is a good idea to keep them in order of dependencies, as it can provide a quick visual reference to the order in which services will be raised.

However, it is important to note that Dependency Injection cannot help us resolve circular references, and they will cause the application to crash.

Demonstration: How to Use Dependency Injection

In this demonstration, you will learn how to create a service, register it in the **ConfigureServices** method and inject it into the **Configure** method by using Dependency Injection.

Demonstration Steps

You will find the steps in the section “Demonstration: How to Use Dependency Injection” on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD03_DEMO.md#demonstration-how-to-use-dependency-injection.

Inject Services to Controllers

One of the places where you benefit the most from Dependency Injection in an ASP.NET Core MVC application is the controllers. In ASP.NET Core MVC, a controller is a class with one or more methods, which are referred to as actions, designed with the intention of receiving requests and formulating a response based on the request.

Controllers will not be deeply covered in this topic, as they are covered in Module 4, "Developing Controllers". Instead, in this topic, you will learn how to create a basic controller, how to send basic content to the response, how to set up the configuration for the ASP.NET Core MVC applications, and finally, how to inject a service and call it from within the controller.

To even start up an ASP.NET Core MVC application, you will first need to update the **ConfigureServices** method with a call to the **AddMvc** method on the **services** parameter. After this is done, the system will load up various services that are required for standard ASP.NET Core MVC behavior. Later, you will need to update the **Configure** method and add a call to the **UseMvcWithDefaultRoute** or **UseMvc** middleware. After you have done this, you will have a running ASP.NET Core MVC environment.

- A controller is used to handle requests from the client
- Controllers support constructor dependency injection
- If the internal behavior of a service or its dependencies change, you will not need to update the controller
- You cannot have more than one constructor in the controller, as the default dependency injection container cannot handle it

 **Note:** Adding MVC to the application can be done inside the **Configure** method by calling either the **UseMvcWithDefaultRoute** or **UseMvc** middleware. The differences between them will be further explained in Module 4, "Developing Controllers". For now, you will use **UseMvcWithDefaultRoute**, as it does not require additional configuration.

The following code demonstrates a basic ASP.NET Core MVC configuration:

ASP.NET Core MVC configuration

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page not found");
    });
}
```

 **Best Practice:** Ideally, the call to **UseMvcWithDefaultRoute** or **UseMvc** will appear as one of the later middleware in the pipeline. This is due to wanting custom behavior and static files to be loaded first and not to be accidentally directed to ASP.NET Core MVC controllers. However, should the need arise, the order may be changed.

Once you have set up the ASP.NET Core MVC environment, you will want to configure a controller for your use. First of all, you should create a folder named **Controllers** inside of the application root. After that, you will want to add a new controller to that folder. Currently, this topic will not be covering how to set up any specific controllers and actions. Instead, you will use a controller called **HomeController** and use the **Index** action. This controller and action will be navigated to by default when you use **UseMvcWithDefaultRoute** middleware.

 **Best Practice:** While not required to be placed inside of a **Controllers** folder, it is generally a good practice to do so. A developer working on controllers, will usually look in the **Controllers** folder and placing them outside of that folder can be a source of confusion.

The following code is an example of a basic controller:

A basic controller

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return Content("Hello from controller");
    }
}
```

In this example, you can see a basic controller. Whenever the user requests an empty relative URL or alternatively requests the relative URL **/home/** or **/home/index**, the **Hello from Controller** string will be displayed in the browser.

 **Note:** The **Content** method allows us to return a string response. Additional methods which return data that implements the **IActionResult** interface will be covered in Module 4, "Developing Controllers".

Once you have the controller set up, you will be able to inject services through the constructor. This behaves in a similar way to Dependency Injection into services, where you can add support for a service by explicitly adding a reference to the interface inside the constructor. Inside the constructor, you will be able to save the instance for use in the specific methods.

The following code is an example of a service that will be injected into the controller:

Creating the service

```
public interface IMyService
{
    string DoSomething();
}
public class MyService : IMyService
{
    public string DoSomething()
    {
        return "something";
    }
}
```

The following code is an example of registering the service:

Startup configuration

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IMyService, MyService>();
    services.AddMvc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page not found");
    });
}
```

The following code is an example of Dependency Injection into a controller:

Controller dependency injection

```
public class HomeController : Controller
{
    private IMyService _myService;

    public HomeController(IMyService myService)
    {
        _myService = myService;
    }

    public IActionResult Index()
    {
        return Content(_myService.DoSomething());
    }
}
```

In this example, you can see a controller. Whenever the user requests an empty relative URL or alternatively requests the relative URL **/home/** or **/home/index**, the **something** string will be displayed in the browser.



Note: The built-in Dependency Injection container in ASP.NET Core does not support multiple constructors. Should the need arise to support multiple constructors alongside Dependency Injection, you will need to use a different dependency injector.

Service Lifetime

Now that you have learned how to create services, the concept of service lifetime will be covered. In general, while running the application, you want to be able to correctly preserve the service state in the application. If you keep all of the services in the application running constantly, you run the risk of creating deadlocks while dealing with external resources as well as potential issues with threading, as each singleton service is locked to one thread since it is first instantiated. On the other hand, keeping all services completely stateless can deprive us of the ability to retain data temporarily and prevent us from being able to manage data correctly.

- **AddSingleton** – Instantiates once in the application's lifetime
- **AddScoped** – Instantiates once per request made to the server
- **AddTransient** – Instantiates every single time the service is injected

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IFirstService, FirstService>();
    services.AddScoped<SecondService, SecondService>();
    services.AddTransient<IThirdService, ThirdService>();
}
```

AddSingleton

To handle persistent data, which you want to persist inside the memory rather than in an external source (such as a database, file, or another source), you will require the service instance to persist throughout the entire application lifespan. This can help you keep the same data consistent and persistent. To handle this case, you will register the service using **AddSingleton**. This tells the Dependency Injection container to create this service once and then to inject the same instance as long as the application remains running. As an important note, should the application be stopped for any reason, this service will be stopped as well.

The following code is an example of a service that generates a random number on instantiation and a wrapper service that utilizes the first service:

Random number service

```
public interface IRandomService
{
    int GetNumber();
}

public class RandomService : IRandomService
{
    private int _randomNumber;

    public RandomService()
    {
        Random random = new Random();
        _randomNumber = random.Next(1000000);
    }
    public int GetNumber()
    {
        return _randomNumber;
    }
}

public interface IRandomWrapper
{
    int GetNumber();
}

public class RandomWrapper : IRandomWrapper
{
    private IRandomService _randomService;

    public RandomWrapper(IRandomService randomService)
    {
```

```

        _randomService = randomService;
    }

    public int GetNumber()
    {
        return _randomService.GetNumber();
    }
}

```

This is an example of a service that generates a random number on instantiation between 0 and 999,999, as well as a service that provides a wrapper service to the random service. The wrapper service is designed with the intention of creating a separate instance of Dependency Injection from the controller. It will be used in all the other examples in this topic.

The following code demonstrates a controller that receives the services by injection:

Injecting services to a controller

```

public class HomeController : Controller
{
    private IRandomService _randomService;
    private IRandomWrapper _randomWrapper;

    public HomeController(IRandomService randomService, IRandomWrapper randomWrapper)
    {
        _randomService = randomService;
        _randomWrapper = randomWrapper;
    }

    public IActionResult Index()
    {
        string result = $"The number from service in controller: { _randomService.GetNumber()}, the number from wrapper service: { _randomWrapper.GetNumber()}";
        return Content(result);
    }
}

```

This is an example of the controller you will use in conjunction with the random service and the wrapper service. It will be used in the other examples in this topic.

The following code is an example of configuring the services as singleton:

AddSingleton usage

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton();
    services.AddSingleton();
    services.AddMvc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page not found");
    });
}

```

In this example, upon a request to the **Index** action the **The number from service in controller: x, the number from wrapper service: x** string will be displayed in the browser. Both the wrapper service and the random service will provide the same value of x. It will remain constant no matter how many times you request the **Index** action. Only by restarting the application will the value of x change.

AddScoped

Sometimes you will want to maintain data throughout the lifetime of a single request without affecting data for other requests. This can be useful for dealing with particular parameters for the specific request such as query string parameters or data which is retrieved due to a specific request, such as user information relating to the user that made the request. For this purpose, you will use **AddScoped**. When a service is registered by using **AddScoped**, all instances where it is injected as a result of the current request being processed will receive the same instance. However, for any other request, a new instance will be created.

The following code is an example of configuring services as scoped:

AddScoped usage

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IRandomService, RandomService>();
    services.AddScoped<IRandomWrapper, RandomWrapper>();
    services.AddMvc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page not found");
    });
}
```

In this example, upon a request to the **Index** action the **The number from service in controller: x, the number from wrapper service: x** string will be displayed in the browser, with the values of both the random service and the wrapper service being identical. The value will change each time you make a request to the **Index** action since the services are instantiated per request.

AddTransient

Some services will end up being stateless and will not need to store data. In that case, you will want to use **AddTransient**. **AddTransient** is instantiated individually every time it's injected, with each instance being completely separate from all others. Due to this, data should never be stored on a transient service.

The following code is an example of configuring services as transient:

AddTransient usage

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IRandomService, RandomService>();
    services.AddTransient<IRandomWrapper, RandomWrapper>();
    services.AddMvc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
```

```
        await context.Response.WriteAsync("Page not found");
    });
}
```

In this example, upon a request to the **Index** action the **The number from service in controller: x, the number from wrapper service: y** string will be displayed in the browser, with the values of x and y being different. These values will change whenever you make a request to the **Index** action, since each time the service will be injected into a new controller instance, and a new wrapper instance.

Lab: Configuring Middleware and Services in ASP.NET Core

Scenario

The Adventure Works company wants to develop a website about ball games. For this, the company needs to perform a survey to determine the popularity of different ball games. As their employee, you are required to create a survey site.

Objectives

After completing this lab, you will be able to:

- Use Microsoft ASP.NET Core static files such as HTML, CSS, and image files.
- Create and use a custom middleware and use its context information.
- Create and use services with ASP.NET Core built-in dependency injection.
- Inject a service to an ASP.NET Core MVC controller.

Lab Setup

Estimated Time: **75 Minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD03_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD03_LAK.md.

Exercise 1: Working with Static Files

Scenario

To create the poll, the application needs a styled HTML page. The HTML page must post the poll results to the server. To transfer the results to the server you will use an HTML form.

The main tasks for this exercise are the following:

1. Create a new project by using the ASP.NET Core Empty project template.
2. Run the application.
3. Add an HTML file to the **wwwroot** folder.
4. Run the application - the content of the HTML file is not displayed.
5. Enable working with static files.
6. Run the application - the content of the HTML file is displayed.
7. Add an HTML file outside of the **wwwroot** folder.
8. Run the application - the content of the HTML file outside the **wwwroot** folder is not displayed.

Exercise 2: Creating Custom Middleware

Scenario

The server needs to handle the client's request. You have been asked to find which ball game was chosen by the user. To do this, you will create a middleware.

The main tasks for this exercise are the follows:

1. Create a middleware.

2. Run the application.
3. Change the order of the middleware.

Exercise 3: Using Dependency Injection

Scenario

You will need to aggregate the votes and store them for future use. You will use services to manage and preserve the data.

The main tasks for this exercise are as follows:

1. Define an interface for a service.
2. Define an implementation for the service.
3. Use dependency injection.
4. Run the application.

Exercise 4: Injecting a Service to a Controller

Scenario

In this exercise, you will create an ASP.NET Core MVC controller to display the poll results.

The main tasks for this exercise are the following:

1. Enable working with MVC.
2. Add a controller.
3. Run the application.
4. Use dependency injection in a controller.
5. Run the application.

Question: What is the difference between `app.Use` and `app.Run` in the `Configure` method in the `Startup` class?

Question: What will change when you update the service configuration in the `ConfigureServices` method from `AddSingleton` to `AddScoped`, or to `AddTransient`?

Question: What happens to the `UseStaticFiles` middleware when the browser is directed to a path where no static file is found?

Module Review and Takeaways

In this module, you learned about how to set up middleware, and how to create and inject services, which are crucial building stones in any ASP.NET Core application. By using middleware, you can interact with the client request, and with the client response. By using Dependency Injection alongside services, you can easily make reusable, non-hard coded components that can be reused and updated without breaking any other code. In Module 4, "Developing Controllers", you will learn how to create controllers, and how to route requests in an ASP.NET Core MVC application.

Review Question

Question: What are the advantages of using Dependency Injection over instantiating references?

Best Practice

Remember that the order of middleware in the **Configure** method is crucial to the application's behavior. While building the middleware pipeline it is important that you think of any potential clashing issues that may occur inside the middleware and design it accordingly.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
While running an application with UseMvc or UseMvcWithDefaultRoute in the Configure method you get an error: System.InvalidOperationException: 'Unable to find the required services...'	Verify that there is a call to AddMvc in the ConfigureServices method.
You navigate to an action in an MVC controller but get an HTTP 500 error page.	Ensure that all services are correctly registered in the ConfigureServices method.

Module 4

Developing Controllers

Contents:

Module Overview	04-1
Lesson 1: Writing Controllers and Actions	04-2
Lesson 2: Configuring Routes	04-10
Lesson 3: Writing Action Filters	04-26
Lab: Developing Controllers	04-32
Module Review and Takeaways	04-34

Module Overview

ASP.NET Core MVC is a framework for building web applications by using the Model-View-Controller (MVC) architectural pattern. The controller is essentially responsible for processing a web request by interacting with the model and then passing the results to the view. The model represents the business layer, sometimes referred to as the domain, and may include data objects, application logic, and business rules. The view uses the data that it receives from the controller to produce the HTML or other output that is sent back to the browser.

In this module, you will learn how to develop controllers. Controllers are central to MVC applications. Understanding how controllers work is crucial to being able to create the appropriate model objects, manipulate them, and pass them to the appropriate views.

A controller is a class. It contains several methods. These methods are called actions. When an MVC application receives a request, it finds which controller and action should handle the request. It determines this by using Uniform Resource Locator (URL) routing.

URL routing is another very important concept necessary for developing MVC applications. The ASP.NET Core MVC framework includes a flexible URL routing system that enables you to define URL mapping rules within your applications.

To maximize the reuse of code in controllers, it is important to know how to write action filters. You can use action filters to run code before or after every action in your web application, on every action in a controller, or on other combinations of controller actions.

Objectives

After completing this module, you will be able to:

- Add a controller to a web application that responds to user actions that are specified in the project design.
- Add routes to the ASP.NET Core routing engine and ensure that URLs are user-friendly in an MVC web application.
- Write code in action filters that runs before or after a controller action.

Lesson 1

Writing Controllers and Actions

A controller is a class that usually derives from the **Controller** base class. Controllers contain methods, known as *actions*, which respond to user requests, process the requests, and return the results to the response stream. The results they return can take several forms, though they are usually represented by objects that implement the **IActionResult** interface. The most common object is a **ViewResult** object that instructs the MVC to render a particular view. However, actions might also yield other types of results, such as string content and instructions to redirect to other actions.

To process incoming user requests, manage user input and interactions, and implement relevant application logic, it is important to know how to create controllers and actions. It is also important to know about the parameters that an action receives.

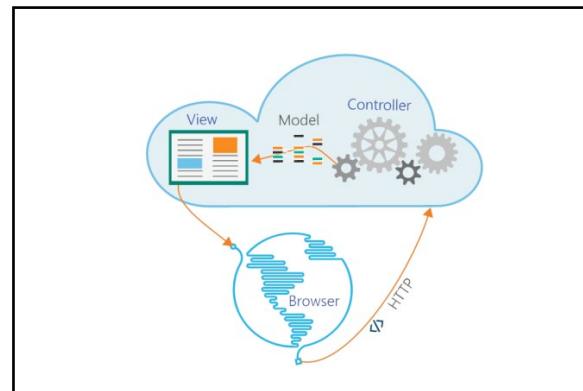
Lesson Objectives

After completing this lesson, you will be able to:

- Describe how a controller responds to user actions in an MVC web application.
- Write controller actions to respond to web browser requests.
- Explain how to pass information to views.
- Create controllers and actions.

Responding to User Requests

When an MVC web application receives a request from a web browser, it instantiates a **Controller** object to respond to the request. Then it determines the action method that should be called in the **Controller** object. The application uses a model binder to determine the values to be passed to the action as parameters. Often, the action creates a new instance of a model class. This model object may be passed to a view to display results to the user. Action methods can do many other things such as rendering views and partial views, and redirecting to other websites



Best Practice: Most simple examples pass model objects to the view. However, for large applications, this is generally not a best practice. For large applications, we recommend that you use ViewModels to separate the presentation from the domain.

The following code shows a simple model class:

A simple model class

```
public class SimpleModel
{
    public string Value { get; set; }
}
```

The following code shows a controller that creates a model and passes it to a view:

A controller that passes a model to a view

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        SimpleModel model = new SimpleModel() { Value = "My Value" };
        return View(model);
    }
}
```

The following code shows a view that receives the model and uses it to generate the response:

A view that uses a model

```
@model MyWebApplication.Models.SimpleModel

 @{
     Layout = null;
 }

 <!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    @Model.Value
</body>
</html>
```

The User Request

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

- **<http://www.contoso.com/>**. This URL is the home page of the website and specifies no further information.
- **<http://www.contoso.com/photo>**. This URL specifies an extra value, **photo**. By default, this value is interpreted as the name of a controller.
- **<http://www.contoso.com/photo/index>**. This URL specifies a second value, **index**. By default, this value is interpreted as the name of an action within the controller.
- **<http://www.contoso.com/photo/display/1>**. This URL specifies a third value, **1**. By default, this value is interpreted as a parameter that should be passed to the **display** action method.

 **Note:** You can modify the preceding behavior in several ways. For example, you can create routes that interpret the preceding URLs differently. These examples are true only when the default route exists. Routes will be covered in Lesson 2, "Configuring Routes".

Let us consider a request whose URL is `http://www.contoso.com/photo`. By default, ASP.NET Core MVC will extract the word **photo** from the URL and instantiate a controller named **PhotoController**, if one exists. You should follow this convention when you create and name controllers. Otherwise, the application will return unexpected 404 errors and the controllers will not work as intended.

The Microsoft Visual Studio project templates include a folder named **Controllers**. Programmers should create their controllers in this folder or in its subfolders. MVC relies on convention over configuration. Therefore, we recommend adhering to the conventions. By default, Visual Studio places controllers in the **ProjectName.Controllers** namespace.

Writing Controller Actions

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

Controller actions are public methods that return an **IActionResult** interface. Therefore, actions can return objects of classes that implement the **IActionResult** interface. For example, you can create an action that calls the **View()** helper method, which creates and returns a **ViewResult** object. The **ViewResult** class inherits from the base **ActionResult** class that implements the **IActionResult** interface.

The following code shows how to return a **ViewResult** object from an action:

An action that returns a **ViewResult** object

```
public ViewResult Index()
{
    return View();
}
```

- An action is a public method of the controller class
- Actions can return objects that implement the **IActionResult** interface

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

For every action that returns a **ViewResult** object, there should be a corresponding view template. The view will generate the output sent to the browser.



Note: Views will be covered in Module 5, "Developing Views".

Actions usually use **ActionResult** as the return type. The **ActionResult** class is an abstract class, and you can use a range of derived classes that inherit from it to return different responses to the web browser. An example of a class that derives from the **ActionResult** class is the **ContentResult** class, which can be used to return text to the web browser. You can return plain text, XML, a comma-separated table, or other text formats. This text can be rendered by the web browser or processed by client-side code.

The following code shows how to return a **ContentResult** object from an action:

An action that returns a ContentResult object

```
public ContentResult Index()
{
    return Content("some text");
}
```

The RedirectToActionResult Class

Another useful return type from an action is **RedirectToActionResult**. The **RedirectToActionResult** class derives from the **ActionResult** class. You can use this action result to return an HTTP 302 response to the browser. This causes the browser to send the request to another action method. To return a **RedirectToActionResult** class from an action, you should use the **RedirectToAction** method. In the parameter of the **RedirectToAction** method, you need to specify the name of the action.

The following code shows how to use the **RedirectToAction** method to redirect to another action in the same controller:

Redirect to another action in the same controller

```
public class HomeController : Controller
{
    public RedirectToActionResult Index()
    {
        return RedirectToAction("AnotherAction");
    }

    public ContentResult AnotherAction()
    {
        return Content("some text");
    }
}
```

The RedirectToRouteResult Class

In addition to the **RedirectToActionResult** object, an action might return a **RedirectToRouteResult** object. The **RedirectToRouteResult** class derives from the **ActionResult** class. You can use this action result to redirect the web browser to another route. In the constructor of the **RedirectToRouteResult** class, you can pass the route. For example, you can pass the name of the controller and the name of the action.

 **Note:** Routes will be covered in Lesson 2, "Configuring Routes".

The following code shows how to use the **RedirectToRouteResult** object to redirect to an action in another controller:

Redirect to an action in another controller

```
public class HomeController : Controller
{
    public RedirectToRouteResult Index()
    {
        return RedirectToRoute(new
        {
            controller = "Another",
            action = "AnotherAction"
        });
    }
}
```

```
public class AnotherController : Controller
{
    public ContentResult AnotherAction()
    {
        return Content("some text");
    }
}
```

The StatusCodeResult Class

An action might also return a **StatusCodeResult** object. The **StatusCodeResult** class provides a way to return an action result with a specific HTTP response status code and description. For example, you might use a **StatusCodeResult** object to return an HTTP 404 Not Found error.

The following code shows how to use the **StatusCodeResult** object to return an HTTP 404 Not Found error:

Return an HTTP 404 Not Found error

```
public class HomeController : Controller
{
    public StatusCodeResult Index()
    {
        return new StatusCodeResult(404);
    }
}
```

Other common action results include:

- **PartialViewResult** You can use this action result to generate a section of an HTML page rather than a complete HTML page. Partial views can be reused in many views throughout a web application.
- **RedirectResult** You can use this action result to redirect to a specific URL, either within your web application or in an external location.
- **FileContentResult** You can use this action result to return a file from an action method.

Using Parameters

When users request webpages, they often specify additional information besides the name of the webpage itself. For example, when they request a product details page, they might specify the name or catalog number of the product to display. Such extra information is referred to as parameters. The powerful model binding mechanism helps make these parameters more flexible and concise.

The model binders obtain parameters from a user request and pass them to action methods.

The model binders can locate parameters in a posted form, routing values, a query string, or posted files. If the model binders find a parameter declared in the action method that matches the name and type of a parameter from the request, the action method is called and the parameter is passed from the request. This arrangement enables you to obtain and use parameters in your actions.

- The model binders obtain parameters from a user request and pass them to action methods
- There are several ways to retrieve parameters, including:
 - The Request property
 - The FormCollection object
 - Routing

For example, assume that an MVC web application contains a route in the **Startup.cs** file. The model binder can use this route to find the parameters and then pass them to an action.

The following code shows a route that appears in the **Startup.cs** file:

Example of a route

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action}/{id?}",
            defaults: new { controller = "Home", action = "Index" });
    });
}
```

The following code shows how the model binder locates parameters in routing values:

The Controller class

```
public class HomeController : Controller
{
    public IActionResult Index(string id)
    {
        return Content(id);
    }
}
```

When a user types the URL <http://host/Home/Index/1>, the **Index** action method of the **HomeController** class is called. The **id** parameter of the method gets the value **1**. The browser displays **1** as the output in this case.

The RouteData Property

Behind the scenes, the model binder uses the **RouteData** property. The **RouteData** property encapsulates the information about the route. It contains a property named **Values**, which can be used to get the value of the parameter.

The following code shows how the model binder locates parameters in routing values by using the **RouteData** property. It has the same output as the code above:

The Controller class

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        string id = (string)RouteData.Values["id"];
        return Content(id);
    }
}
```

In a similar way, the model binder locates parameters in a query string. For example, in case a user types the URL <http://host/Home/Index?title=someTitle>, the **Index** action method of the **HomeController** class is called. The **title** parameter of the method gets the value **someTitle**. The browser displays **someTitle** as the output in this case.

The following code shows how the model binder locates parameters in a query string:

The Controller class

```
public class HomeController : Controller
{
    public IActionResult Index(string title)
    {
        return Content(title);
    }
}
```

Using ViewBag and ViewData to Pass Information to Views

You can pass a model object from the action method to the view by using the **View()** method. This method is frequently used to pass information from a controller action to a view. This approach is recommended because it adheres closely to the MVC pattern, in which each view renders the properties found in the model class that it receives from the controller. You should use this approach wherever possible.

 **Note:** Models will be covered in Module 6, "Developing Models".

- Models are the best way to pass data from controllers to views
- In some cases, you may want to augment the model with additional data without modifying it



However, in some cases you might want to augment the information in the model class with some extra values. For example, you might want to send a title, which needs to be inserted in the page header, to the view. Furthermore, some views do not use model classes. The home page of a website, for example, often does not have a specific model class. To help in these situations, you can use two other mechanisms to provide extra data: the **ViewBag** and **ViewData** properties. These are almost identical.

Using the ViewBag Property

The **ViewBag** property is a dynamic object that is part of the base **Controller** class. Because it is a dynamic object, you can add to it values of any type in the action method. In the view, you can use the **ViewBag** property to obtain the values added in the action.

The following code shows how to add properties to the **ViewBag** property:

Adding properties to the ViewBag property

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        ViewBag.Message = "some text";
        ViewBag.ServerTime = DateTime.Now;
        return View();
    }
}
```

The following code shows how to obtain and use the same properties in a view by using Razor:

Using the ViewBag properties

```
<p>
    Message is: @ViewBag.Message
</p>
<p>
    Server time is: @ViewBag.ServerTime.ToString()
</p>
```

 **Note:** Razor will be covered in Module 5, "Developing Views".

Using the ViewData Property

You can pass extra data to views by using the **ViewData** property. This feature is available for developers who prefer to use dictionary objects. In fact, **ViewBag** is a dynamic wrapper above the **ViewData** dictionary. This means that you could assign a value in a controller action by using **ViewBag** and read the same value in the view by using **ViewData**.

In action methods, you can add data to the **ViewData** property by using key/value pairs as shown in the following lines of code:

Adding data to the ViewData property

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        ViewData["Message"] = "some text";
        ViewData["ServerTime"] = DateTime.Now;
        return View();
    }
}
```

To obtain and use the same values in a view, you can use the following Razor code:

Using ViewData values

```
<p>
    Message is: @ViewData["Message"]
</p>
<p>
    Server time is: @((DateTime)ViewData["ServerTime"])
</p>
```

Demonstration: How to Write Controllers and Actions

In this demonstration, you will learn how to add a controller to an MVC application and explore different ways to pass information to the view.

Demonstration Steps

You will find the steps in the section "Demonstration: How to Write Controllers and Actions" on the following page: https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD04_DEMO.md#demonstration-how-to-write-controllers-and-actions.

Lesson 2

Configuring Routes

You can use ASP.NET Core to control the URLs that a web application uses. You can link the URLs and the content by configuring routes. A route is a rule. Because routes are configurable, you can link URLs and content more effectively. ASP.NET Core uses routes to parse a requested URL to determine the controller and action to which the request must be forwarded.

You need to know how to add a new route to your application. You also need to know how the routing engine interprets a route so that a requested URL is handled by the appropriate controller and action, and your application can expose meaningful URLs.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the features of the ASP.NET Core routing engine.
- Describe the benefits of adding routes to an ASP.NET Core MVC web application.
- Explain how to add and configure routes by using convention-based routing.
- Explain how to use routes to pass parameters.
- Explain how to add and configure routes by using attributes.
- Add routes to manage URL formulation.

The ASP.NET Core Routing Engine

Routing is used to determine the controller class that should handle a request. Routing is also used to decide which action should be called and what parameters need to be passed to that action. The following steps occur when a request is received from a web browser:

1. A controller is instantiated to respond to the request. Routing is used to determine the right controller class to use.
2. The request URL is examined. Routing is used to determine the action that needs to be called in the controller object.
3. Model binding is used to determine the values that should be passed to the action as parameters. The model binding mechanism consults the routing entries to determine if any segments of the URL should be passed as parameters. The model binding mechanism can also pass parameters from a posted form, from the URL query string, or from uploaded files.
4. The action is then invoked. Often, the action creates a new instance of a model class, perhaps by querying the database with the parameters passed to it by the invoker. This model object is passed to a view to render the results and send them to the response.

- Routing determines which controller and action should be called to handle a request
- Routing can be configured centrally in the Startup.cs file and locally by using attributes



Routing governs the way URLs are formulated and how they correspond to controllers and actions. Routing does not operate on the protocol, server, domain, or port number of a URL, but it operates only on the directories and file name in the relative URL. For example, in the URL, <http://www.contoso.com/photo/display/1>, routing operates on the **/photo/display/1** relative path.

In ASP.NET Core, routes are used for two purposes:

- To parse the URLs requested by browsers. This analysis ensures that requests are forwarded to the right controllers and actions. These are called incoming URLs.
- To formulate URLs in webpage links and other elements. When you use helpers such as **Html.ActionLink** in the MVC views, the helpers construct URLs according to the routes in the routing table. These are called outgoing URLs.

 **Note:** Helpers will be covered in Module 5, "Developing Views".

ASP.NET Core MVC allows us to configure routes in two different ways:

- Configure routes by using convention-based routing. In this case, the routes are configured in the **Startup.cs** file and have an impact on the entire application.
- Configure routes by using attributes. As the name implies, attribute routing uses attributes to define routes.

You can combine convention-based routing and attribute-based routing in the same MVC application.

The Default Route

You can configure an ASP.NET Core web application to use a default central route. This route should be located in the **Startup.cs** file. You configure this route by using the **UseMvcWithDefaultRoute** method. This method adds MVC to the **IApplicationBuilder** request execution pipeline with a default route named **default** and the **{controller=Home}/{action=Index}/{id?}** template.

The following code shows how the default route is implemented in the **Startup.cs** file:

The default route

```
public class Startup
{
    // Code is omitted

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        // Code is omitted

        // Add MVC to the request pipeline.
        app.UseMvcWithDefaultRoute();
    }
}
```

The default route is simple and logical. It works well with many web applications. The default route examines the first three segments of the URL. Each segment is delimited by a forward slash:

- The first segment is interpreted as the name of the controller. The routing engine forwards the request to this controller. If a first segment is not specified, the default route forwards the request to a controller called **HomeController**.
- The second segment is interpreted as the name of the action. The routing engine forwards the request to this action. If a second segment is not specified, the default route forwards the request to an action called **Index**.
- The third segment is interpreted as an id value, which is passed to the action as a parameter. This parameter is optional, so if a third segment is not specified, no default value is passed to the action.

For example, if a user requests the URL, <http://www.contoso.com/photo/display/1>, the id parameter **1** is passed to the **Display** action of the **PhotoController** controller.

Discussion: Why Add Routes?

Developers add their own custom routes to a web application for two main reasons:

- To make URLs easier for site visitors to understand. A URL such as <http://www.contoso.com/photo/display/1> is logical to a visitor. However, it requires some knowledge to formulate such URLs. In this case, to type the right URL in the address bar of a web browser, the user must know some information. This information includes the controller name, **PhotoController**, the action name, **Display**, and the **ID** of the required photo. If you use Globally Unique Identifiers (GUIDs) in the database, the **ID** segment of the URL can be long and difficult to remember. Ideally, you should consider what users know and create routes that accept that information. In this example, users may know the title of a photo that they had seen earlier in your application. You should create a route that can interpret a URL such as <http://www.contoso.com/photo/title/My%20Photo%20Title> and forward the URL to an appropriate action to display the right photo. Although users usually click links to make requests, friendly URLs like these make a site easier to use.
- To improve search engine rankings. Search engines do not prioritize webpages that have GUIDs or long query text in the URL. Some web bots do not even crawl such links. In addition, some search engines boost a page's ranking when one or more of its keywords appear in the URL. User-friendly URLs are therefore a critical tool in Search Engine Optimization (SEO).

Why add routes?

- To make URLs easier for site visitors to understand
- To improve search engine rankings

What Is Search Engine Optimization?

Most users find web applications by using search engines. Users tend to visit the links that appear at the top of the first page of search engine results more frequently than those that appear at the bottom of the page or on the later pages. For this reason, website administrators and developers try to ensure that their web application appears at the top of the search engine results by using a process known as SEO. SEO ensures that more people visit your web application.

Search engines examine the content of your web application by crawling it with a program called a web bot. If you understand the priorities that web bot and search engine indexes use to order search results, you can create a web application that conforms to those priorities and thereby appears high in search engine results.

Various search engines have different web bots with different algorithms to prioritize results. The complete details of these algorithms are not usually published.

You can research for the best practices, but you should not abuse them because search engines penalize abuse.

- Most users find web applications by using search engines
- It is important to ensure that your web application appears at the top of the search engine results



Information architecture is a subject that is closely related to SEO. This is because both information architecture and SEO are relevant to the structure, hierarchy, and accessibility of the objects in your web application. Users click links on menus to navigate to the pages that interest them. Web bots use the same links to navigate the web application and crawl its content. Users prefer URLs without GUIDs and long query text because they are meaningful. Web bots often ignore links with GUIDs and long query text in them. In addition, when keywords appear in URLs, web bots prioritize a webpage in search results.

As an ASP.NET Core MVC developer, you must understand SEO principles and use them whenever you write code to ensure that you do not impact the search engine positioning of your web application. Routes and the configuration of the routing engine are also critical, because by using routes, you can control the URLs that your web application generates.

Configuring Routes by Using Convention-Based Routing

You can use convention-based routing by defining routes in your project's **Startup** class located in the file **Startup.cs**.

The following code shows a **Startup** class that is configured to use MVC:

The Startup class

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseMvc();
    }
}
```

- Convention-based routes might contain the following properties: name, template, defaults, constraints and dataTokens

- Custom routes can be added as shown below:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{param}",
        defaults: new { controller = "Some", action = "ShowParam" },
        constraints: new { param = "[0-9]+" });
});
```

- Routes should be added in the appropriate order

The **ConfigureServices** method is used to add services to the container. The **Configure** method is used to add MVC to the request pipeline. As you can see, no routes have been configured in the **Startup** class yet.

 **Note:** The **Startup** class was covered in Module 3, "Configure Middleware and Services in ASP.NET Core".

Assume that there is a controller in the MVC application. Will it be possible to navigate to the controller?

The following code shows a **Controller** class:

The Controller class

```
public class SomeController : Controller
{
    public IActionResult Display()
    {
        return Content("Reached the action");
    }
}
```

```
}
```

If a user requests the `/` relative URL, the request will fail. Also, if a user requests the `/Some/Display` relative URL, the request will fail.

The reason for this is that no routes are configured in the `Startup` class. In this case, you must add a convention-based route. Before adding a route, it is important to understand the properties of a route. This is to ensure that these properties match the URL segments correctly and pass the requests and parameters to the right location.

Properties of a Route

The properties of a route include `name` and `template`, both of which are string properties. The `name` property assigns a name to a route. It is not involved in matching or request forwarding. The `template` property is a URL pattern that is compared to a request URL to determine if the route should be used. You can use segment variables to match a part of the URL. Use braces to specify a segment variable.

For example, if you set the template of a route to `{controller}/{action}` and then a user requests the `/Some/Display` relative URL, the request will succeed. The `{controller}` segment variable maps to `SomeController`, and the `{action}` segment variable maps to `Display`. If a user requests the `/` relative URL, the request will fail because no default values were set.

The following code shows the `Configure` method in the `Startup` class with a route:

The Configure method

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action}");
    });
}
```

You can change the value of the `name` property. For example, you can change it to `MyRoute`, as long as the `name` property is unique in the `routes` collection. You also have the flexibility to change the order of the segment variables. If you change the route that is written in the template to `{action}/{controller}` and then a user requests the `/Some/Display` relative URL, the request will fail. However, if a user requests the `/Display/Some` relative URL, the request will succeed.

The following code shows a route with different segment variables order and a different name:

The Configure method

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "myRoute",
            template: "{action}/{controller}");
    });
}
```

If you use the `{controller}/{action}` template, only URLs with two segments will be considered legal. For example, if a user requests the `/Some/Display/1` relative URL, the request will fail.

Adding Segments to a Template

To support additional segments, you should update the template accordingly. For example, you can change the value of the template to **{controller}/{action}/{param}**. In this case, if a user requests the relative URL **/Some/Display**, the request will fail and if the user requests the **/Some/Display/1** relative URL, it will succeed.

The following code shows how an additional segment named **param** can be added to a route:

The Configure method

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action}/{param}");
    });
}
```

Furthermore, assume that there is an action that gets a parameter named **param**. In this case, if there is a route with the template **{controller}/{action}/{param}**, the model binding mechanism will map the value entered in the URL to the value of the **param** parameter in the action. For example, assume that there is an action named **ShowParam** that gets a parameter named **param** and sends its value to the browser. In this case, if a user requests the **/Some>ShowParam/hello** relative URL, the **param** parameter gets the value **hello**, which in turn is sent to the browser.

The following code shows an action that gets a parameter named **param** and sends its value to the browser:

The ShowParam action

```
public class SomeController : Controller
{
    public IActionResult ShowParam(string param)
    {
        return Content(param);
    }
}
```

The defaults Property

The **defaults** property is another important property of a route. This property can assign default values to the segment variables in the URL pattern. Default values are used for segment variables when the request does not specify them.

For example, refer to the following code that shows a route with the **defaults** property:

Route with defaults property

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{param}",
        defaults: new { controller = "Some", action = "ShowParam", param = "val" });
});
```

If you use the above route, the **ShowParam** action of the **SomeController** controller will be invoked when a user requests the **/**, **/Some**, and **/Some>ShowParam** relative URLs. In all these cases, the value of the **param** parameter will be **val**.

You can set the default values of a route directly in the **template** property.

For example, you can replace the **template** and **default** properties of the previous example with a **template** property with the following value: **{controller=Some}/{action>ShowParam}/{param=val}**. These two examples create equivalent routes:

Route with default values in the template property

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Some}/{action>ShowParam}/{param=val}");
});
```

The constraints Property

Sometimes, you must place extra constraints on the route to ensure that it only matches specific requests. The **constraints** property enables you to specify a regular expression for each segment variable. The route will match a request only if all the segment variables match the regular expressions that you specify.

For example, assume that you want to ensure that the **param** segment includes only digits. For this, you can set the route with the **constraints** property:

A route with the constraints property

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{param}",
        defaults: new { controller = "Some", action = "ShowParam" },
        constraints: new { param = "[0-9]+" });
});
```

If a user requests the **/Some>ShowParam/1** relative URL, the request will succeed. However, if a user requests the **/Some>ShowParam/a** relative URL, the request will fail.

It is possible to specify route constraints directly in the **template** property.

For example, assume that you want to ensure that the route value **param** must be convertible to an integer. For this, you can set the following route:

A route with constraints in the template property

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Some}/{action>ShowParam}/{param:int}");
});
```

The dataTokens Property

Another property you can set in a route is the **dataTokens** property. The **dataTokens** property enables you to specify the data tokens for the route. Each data token contains a name and a value.

For example, refer to the following code that shows a route with the **dataTokens** property:

A route with the **dataTokens** property

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{param}",
        defaults: new { controller = "Some", action = "ShowParam" },
        constraints: new { param = "[0-9]+" },
        dataTokens: new { locale = "en-US" });
});
```

Order of Evaluation of Routes

Routes are evaluated in the order in which they are added. If a route matches a request URL, it is used. If a route does not match, it is ignored and the next route is evaluated. For this reason, you should add routes in the appropriate order. You should add the most specific routes first, such as the routes that have no segment variables and no default values. Also, routes with constraints should be added before routes without constraints.

The following code shows how two routes can be added:

Adding two routes

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "firstRoute",
            template: "{controller}/{action}/{param}",
            defaults: new { controller = "Some", action = "ShowParam" },
            constraints: new { param = "[0-9]+" });

        routes.MapRoute(
            name: "secondRoute",
            template: "{controller}/{action}/{id?}",
            defaults: new { controller = "Home", action = "Index" });
    });
}
```

Using Routes to Pass Parameters

The routing engine separates the relative URL in a request into one or more segments. Each forward slash delimits one segment from the next. If you want one of the segments to specify the controller's name, you can use the **{controller}** segment variable. This variable is interpreted as the controller to be instantiated. Alternatively, to fix a route to a single controller, you can specify a value for the **controller** variable in the **defaults** property. In a similar manner, if you want one of the segments to specify the action method, you can use the

{action} segment variable. The action invoker always interprets this variable as the action to call.

You can access the values of segment variables by using one of two methods:

- Using the **RouteData.Values** collection


```
public IActionResult Print()
{
    string id = (string)RouteData.Values["id"];
    return Content("id: " + id);
}
```
- Using model binding to pass appropriate parameters to actions


```
public IActionResult Print(string id)
{
    return Content("id: " + id);
}
```

Alternatively, to fix a route to a single action, you can specify a value for the **action** variable in the **defaults** property.

Segment variables or default values with other names have no special meaning to MVC and are passed to actions. You can access the values of these variables by using one of two methods: by using the **RouteData.Values** collection or by using model binding to pass values to action parameters.

Using the **RouteData.Values** Collection

In the action method, you can access the values of any segment variable by using the **RouteData.Values** collection. For example, you can access the **controller** class name by using the **RouteData.Values["controller"]** statement, and you can access the **action** method name by using **RouteData.Values["action"]** statement.

The following code shows an action that uses the **RouteData.Values** collection:

The **RouteData.Values** example

```
public class ExampleController : Controller
{
    public IActionResult Print()
    {
        string controller = (string)RouteData.Values["Controller"];
        string action = (string)RouteData.Values["Action"];
        return Content("Controller: " + controller + ". Action: " + action);
    }
}
```

The following code shows a route:

Example of a route

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "someRoute",
        template: "{controller}/{action}/{id?}",
        defaults: new { controller = "Example", action = "Print" });
});
```

If a user requests the **/Example/Print** relative URL, the **Controller: Example. Action: Print** string is displayed in the browser.

In a similar way, you can use the **RouteData.Values** collection to access other segment variables.

The following code shows a controller that uses the **RouteData.Values** collection to access the **id** segment value:

Using the **RouteData.Values** Collection

```
public class ExampleController : Controller
{
    public IActionResult Print()
    {
        string id = (string)RouteData.Values["id"];
        return Content("id: " + id);
    }
}
```

If a user requests the **/Example/Print/1** relative URL, the following string appears on the screen: **id: 1**.

The question mark in {id?} specifies that **id** is optional. If the relative URL has three segments, the route matches and the third segment is passed to the **RouteData.Values["id"]** value. If the relative URL has only two segments, the route matches as well, because the **{id}** is optional. In this case, the **RouteData.Values["id"]** value is null, and the string **id**: displays on the screen.

Using Model Binding to Obtain Parameters

The default MVC action invoker passes the appropriate parameters to actions. For this it examines the definition of the action to which it must pass parameters. The model binding mechanism searches for values in the request, which can be passed as parameters by name. One of the locations that it searches is the **RouteData.Values** collection. Therefore, you can use the default model binder to pass parameters to action. If the name of an action method parameter matches the name of a route segment variable, the model binder passes the parameter automatically.

The following code shows a controller that uses model binding to access the **id** segment value:

Using model binding

```
public class ExampleController : Controller
{
    public IActionResult Print(string id)
    {
        return Content("id: " + id);
    }
}
```

If a user requests the **/Example/Print/1** relative URL, the action returns the following string: **id: 1**.

The parameters of the **action** method don't have to be of the **string** type. For example, you can change the **Print** action to get an **int** as a parameter.

The following code shows an action that gets an **int** as a parameter:

A model binding example

```
public class ExampleController : Controller
{
    public IActionResult Print(int id)
    {
        return Content("id: " + id);
    }
}
```

If a user requests the **/Example/Print/1** relative URL, the string **id: 1** will be returned. If a user requests the **/Example/Print/a** relative URL, the string **id: 0** will be returned. If a user requests the **/Example/Print** relative URL, the string **id: 0** will be returned. This is because **int** is a value type, which cannot store **null** values.

You can allow value types to have **null** values if you mark them as **Nullable**. For example, if you change the **Print** action to get a parameter of type **int?** and then a user requests the **/Example/Print/a** relative URL, the **null** value will be assigned to **id**.

The following code shows an action that gets a parameter of type **int?**:

A nullable parameter

```
public class ExampleController : Controller
{
    public IActionResult Print(int? id)
    {
        return Content("id: " + id);
    }
}
```

You also can allow a parameter in an action to have optional values.

The following code shows an action that gets a parameter with an optional value:

The optional parameter

```
public class ExampleController : Controller
{
    public IActionResult Print(int id = 444)
    {
        return Content("id: " + id);
    }
}
```

If a user requests the **/Example/Print/a** and **/Example/Print** relative URLs, the string **id: 444** will be returned. If a user requests the **/Example/Print/1** relative URL, the string **id: 1** will be returned.

You can set several segments in the template of a route, and all of them can be matched to parameters of an action.

The following code shows a template with several segments:

Route with several segments

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "someRoute",
        template: "{controller}/{action}/{id}/{title?}",
        defaults: new { controller = "Example", action = "Print" });
});
```

By using the above route, you can write an action that gets two parameters, **id** and **title**. Both parameters can be matched with the segments in the request URL.

The following code shows an action that has two parameters:

An action with two parameters

```
public class ExampleController : Controller
{
    public IActionResult Print(string id, string title)
    {
        return Content("id: " + id + ". title: " + title);
    }
}
```

If a user requests the **/Example/Print** relative URL, the request will fail because the **id** segment in the route is mandatory. If a user requests the **/Example/Print/1** relative URL, the request will succeed because the **title** segment in the route is optional. In this case, the **id** parameter will have the value **1** and the **title** parameter will have the value **null**. If a user requests the **/Example/Print/1/SomeTitle** relative URL, the request will succeed. In this case, the **id** parameter will have the value **1** and the **title** parameter will have the value **SomeTitle**.

Configuring Routes by Using Attributes

ASP.NET Core MVC supports two types of routing, convention-based routing and attribute-based routing. As you have seen earlier, routes in convention-based routing are configured in a centralized place – the **Startup** class that is defined in the **Startup.cs** file. Attribute-based routing allows you to configure routes by using attributes. The primary advantage of attribute-based routing is that it allows you to define your routes in the same file as the controller that the routes refer to.

The following code shows a **Startup** class that is configured to use MVC with no centralized routes:

The Startup class

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseMvc();
    }
}
```

You can use the **Route** attribute to specify the route for a specific action. For example, consider a controller named **MyController** that contains an action named **SomeMethod**. If a user requests the **/Some** relative URL, the request will fail. However, if you add **[Route("Some")]** above the declaration of the **SomeMethod** action, the request will succeed.

The following code shows how to configure a route by using an attribute:

Attribute-based routing

```
public class MyController : Controller
{
    [Route("Some")]
    public IActionResult SomeMethod()
    {
        return Content("Some method");
    }
}
```

Pass Parameters Using Attribute Routing

You can use attribute routing to pass parameters to an action. For example, if you want the **SomeMethod** action to get a parameter named **param**, you can set an attribute to the action in the following way: **[Route("My/{param}")]**. After setting the attribute, if a user requests the **/My/val** relative URL, the request will succeed and **param** will have the value **val**.

- Attribute-based routing allows you to configure routes using attributes
 - It allows you to define your routes in the same file as the controller that they refer to
- ```
[Route("Some")]
public IActionResult SomeMethod()
{
 return Content("Some method");
}
```
- Convention-based routing and attribute-based routing can be used in the same application

The following code shows how to configure an attribute route with a parameter. Note that the components of the route are named differently from the name of the action method:

#### An attribute route with parameter

```
public class MyController : Controller
{
 [Route("My/{param}")]
 public IActionResult SomeMethod(string param)
 {
 return Content(param);
 }
}
```

The **Route** attribute can be used to get several segment variables. The **Route** attribute can also be used to ensure that a segment variable is of a specific type. For example, setting the attribute **[Route("My/{param1}/{param2:int}")]** configures a route in which two segment variables should exist, **param1** and **param2**. **param2** must be of type **int**.

The following code shows how to configure an attribute route with two parameters, where the second parameter must be of type **int**:

#### An attribute route with two parameters

```
public class MyController : Controller
{
 [Route("My/{param1}/{param2:int}")]
 public IActionResult SomeMethod(string param1, int param2)
 {
 return Content("param1: " + param1 + ", param2: " + param2);
 }
}
```

Based on the above example, if a user requests the **/My/a** relative URL, the request will fail. Also, if a user requests the **My/a/b** relative URL, the request will fail because **b** cannot be converted to an **int**. However, if a user requests the **/My/a/1** relative URL, the request will succeed, and **param1** will have the value **a** and **param2** will have the value **1**.

#### Optional Parameters

You can configure a segment variable to be optional by using a question mark. For example, to specify that **param2** should be optional, use **{param2?}**.

The following code shows how to configure an attribute route to be optional:

#### An attribute route with an optional parameter

```
public class MyController : Controller
{
 [Route("My/{param1}/{param2?}")]
 public IActionResult SomeMethod(string param1, string param2)
 {
 return Content("param1: " + param1 + ", param2: " + param2);
 }
}
```

Based on the above example, if a user requests the **My/a** relative URL, the request will succeed because **param2** is optional. In this case, the value of **param2** will be **null**. On the other hand, if a user requests the **My** relative URL, the request will fail because **param1** is not optional.

## Annotate a Controller Class with a Route Attribute

Typically, all the routes in a **controller** class start with the same prefix.

The following code shows two routes with the same prefix:

### Two routes with the same prefix

```
public class MyController : Controller
{
 [Route("My/Method1")]
 public IActionResult Method1()
 {
 return Content("Method1");
 }

 [Route("My/Method2")]
 public IActionResult Method2()
 {
 return Content("Method2");
 }
}
```



**Best Practice:** All the routes in a **controller** class should start with the same prefix.

You can set a common prefix for an entire controller class by using the **[Route]** attribute above the controller class.

The following code shows how a **Route** attribute can be applied to a controller class:

### A Route attribute on a Controller class

```
[Route("My")]
public class MyController : Controller
{
 [Route("Method1")]
 public IActionResult Method1()
 {
 return Content("Method1");
 }

 [Route("Method2")]
 public IActionResult Method2()
 {
 return Content("Method2");
 }
}
```

Based on the above example, if a user requests the **/My/Method1** relative URL, the request will succeed. However, if a user requests the **/Method1** relative URL, the request will fail.

### Combine Convention-Based Routing with Attribute-Based Routing

Convention-based routing and attribute-based routing can be used in the same application.

The following code shows a convention-based route:

### Convention-based route

```
app.UseMvc(routes =>
{
 routes.MapRoute(
 name: "default",
 template: "{controller}/{action}");
```

```
});
```

Consider a controller class named **MyController** that has two actions, an action named **Method1** that has the **[Route("SomeRoute")]** attribute route and an action named **Method2** with no **Route** attribute.

The following code shows a controller class with two actions, one with a **Route** attribute and the second without a **Route** attribute:

### The Controller class

```
public class MyController : Controller
{
 [Route("SomeRoute")]
 public IActionResult Method1()
 {
 return Content("Method1");
 }

 public IActionResult Method2()
 {
 return Content("Method2");
 }
}
```

When a user requests the **/SomeRoute** relative URL, the **Method1** action runs. This is because of the attribute route that is configured above the method. When a user requests the **/My/Method2** relative URL, the **Method2** action runs. This is because of the centralized route that is configured in the **Startup** class.

### Attribute Routing with **Http[Verb]** Attributes

In attribute routing you can use the **Http[Verb]** attributes instead of using the **Route** attribute. The **Http[Verb]** attributes should be used when you want to limit access to the action to a specific HTTP verb. For example, if you want an action to run only when the HTTP verb is **GET**, you can use the **HttpGet** attribute. Similarly, if you want an action to run only when the HTTP verb is **POST**, you can use the **HttpPost** attribute.

The following code shows how a **HttpGet** attribute can be applied to an action method:

### Using the **HttpGet** attribute

```
public class CitiesController : Controller
{
 [HttpGet("/cities/{id}")]
 public IActionResult GetCity(int id) { ... }
}
```

When a user requests the **/cities/1** relative URL, the **GetCity** action runs. This action only matches the HTTP **GET** verb because it is decorated with the **HttpGet** attribute.

To make the attribute routing less repetitive, you can apply the **Route** attribute to a controller, and apply the **Http[Verb]** attribute to an action. In such cases, the **Route** attribute is used to set the common prefix for an entire controller class, and its template is prepended to the template of the action.

The following code shows how you can combine the **Route** attribute with the **Http[Verb]** attributes:

### Combine the **Route** attribute with the **Http[Verb]** attributes

```
[Route("cities")]
public class CitiesController : Controller
{
 [HttpGet]
 public IActionResult ListCities() { ... }
```

```
[HttpGet("{id}")]
public ActionResult GetCity(int id) { ... }
}
```

When a user requests the **/cities** relative URL, the **ListCities** action runs. When a user requests the **/cities/1** relative URL, the **GetCity** action runs.

## Demonstration: How to Add Routes

In this demonstration, you will learn how to configure routes by using convention-based routing and by using attribute-based routing.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Add Routes” on the following page:

[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD04\\_DEMO.md#demonstration-how-to-add-routes](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD04_DEMO.md#demonstration-how-to-add-routes).

## Lesson 3

# Writing Action Filters

In some situations, you might have to run specific code before or after controller actions are run. For example, before a user runs any action that modifies data, you might want to run code that checks the details of the user account. If you add such code to the actions themselves, you will have to duplicate the code in all the actions where you want the code to run. Action filters provide a convenient way to avoid code duplication. You also can use action filters to write thinner controllers and separate responsibilities.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe action filters.
- Create action filters.
- Determine when to use action filters.

### What Are Action Filters?

The MVC programming model enforces the separation of concerns. For example, the business logic in model classes is separate from the input logic in controllers and the user interface logic in views. Each model class is also clearly distinct from other model classes. However, there are scenarios where requirements may be relevant to many parts of your application and cut across logical boundaries. For example, authorization must be applied to many sensitive actions and controllers, regardless of the model and views that they use. These types of requirements are known as cross-cutting concerns. Some common examples of cross-cutting concerns include authorization, logging and caching.

- Filters are MVC classes that can be used to manage cross-cutting concerns in your web application



- You can apply a filter to an action by annotating the action method with an appropriate attribute

Filters are MVC classes that you can use to manage cross-cutting concerns in your web application. You can apply a filter to a controller action by annotating the action method with the appropriate attribute. You can also apply a filter to every action in a controller by annotating the controller class with the attribute.

Filters run after an action is chosen to run. They run within a *filter pipeline*. The stage in the filter pipeline in which the filter is run is determined by the type of the filter. The following are examples of filter types:

- **Authorization filters.** Authorization filters run before any other filter and before the code in the action method. They are used to check a user's access rights for the action.
- **Resource Filters.** Resource filters appear in the filter pipeline after the authorization filters and before any other filters. They can be used for performance reasons. Resource filters implement either the **IResourceFilter** interface or the **IAsyncResourceFilter** interface.
- **Action Filters.** Action filters run before and after the code in the action method. You can use action filters to manipulate the parameters that an action receives. They can also be used to manipulate the result returned from an action. Action filters implement either the **IActionFilter** interface or the **IAsyncActionFilter** interface.

- **Exception Filters.** Exception filters run only if an action method or another filter throws an exception. These filter classes are used to handle errors. Exception filters implement either the **IExceptionFilter** interface or the **IAsyncExceptionFilter** interface.
- **Result Filters.** Result filters run before and after an action result is run. Result filters implement either the **IResultFilter** interface or the **IAsyncResultFilter** interface.

You can create your own filter classes or use existing filter classes. The following are examples of existing filter classes that you can use:

- The **ResponseCacheAttribute** filter class. Annotating an action with the **ResponseCache** attribute will cache the response to the client. The **ResponseCache** attribute contains several properties. One of them is the **Duration** property, which gets or sets the duration, in seconds, for which the response is cached. Caching will be covered in detail in Module 12, "Performance and Communication".
- The **AllowAnonymousAttribute** filter class. Annotating an action with the **AllowAnonymous** attribute will allow users to access an action without logging in. Authentication and authorization will be covered in Module 11, "Managing Security".
- The **ValidateAntiForgeryTokenAttribute** filter class. Annotating an action with the **ValidateAntiForgeryToken** attribute will help prevent cross-site request forgery attacks. Defending from attacks will be covered in Module 11, "Managing Security".

## Creating and Using Action Filters

If you have a cross-cutting concern in your web application, you can create a custom action filter or a custom result filter. You can create custom filters by implementing the **IActionFilter** interface or the **IResultFilter** interface. However, the **ActionFilterAttribute** base class implements both the **IActionFilter** and **IResultFilter** interfaces for you. By deriving your filter from the **ActionFilterAttribute** class, you can create a single filter that can run code both before and after an action is run, and both before and after the result is returned.

### Sample Action Filter

```
public class SimpleActionFilter : ActionFilterAttribute {
 public override void OnActionExecuting(ActionExecutingContext filterContext) {
 Debug.WriteLine("This Event Fired: OnActionExecuting");
 }

 public override void OnActionExecuted(ActionExecutedContext filterContext) {
 Debug.WriteLine("This Event Fired: OnActionExecuted");
 }
}
```

The following code shows how an action filter is used to write text to the Visual Studio Output Window:

### A simple custom action filter

```
public class SimpleActionFilter : ActionFilterAttribute
{
 public override void OnActionExecuting(ActionExecutingContext filterContext)
 {
 Debug.WriteLine("This Event Fired: OnActionExecuting");
 }

 public override void OnActionExecuted(ActionExecutedContext filterContext)
 {
 Debug.WriteLine("This Event Fired: OnActionExecuted");
 }

 public override void OnResultExecuting(ResultExecutingContext filterContext)
 {
 Debug.WriteLine("This Event Fired: OnResultExecuting");
 }
}
```

```
public override void OnResultExecuted(ResultExecutedContext filterContext)
{
 Debug.WriteLine("This Event Fired: OnResultExecuted");
}
```

## Using a Custom Action Filter

After you create a custom action filter, you can apply it to any action method or class in your web application by annotating the method or class with the action filter name.

In the following lines of code, the **SimpleActionFilter** attribute is applied to the **Index** action of the **MyController** controller:

### Using a custom action filter

```
public class MyController : Controller
{
 [SimpleActionFilter]
 public IActionResult Index()
 {
 return Content("some text");
 }
}
```

## Action Filter Context

The **OnActionExecuting**, **OnActionExecuted**, **OnResultExecuting** and **OnResultExecuted** event handlers take a context object as a parameter. The context object inherits from the **FilterContext** class. The **FilterContext** class contains properties that you can use to get additional information regarding the action and the result. For example, you can retrieve the action name by using the **ActionDescriptor.RouteValues** property.

The following code shows how the action name can be retrieved in the **OnActionExecuting** and **OnActionExecuted** event handlers:

### The OnActionExecuting and OnActionExecuted event handlers

```
public class SimpleActionFilter : ActionFilterAttribute
{
 public override void OnActionExecuting(ActionExecutingContext filterContext)
 {
 string actionPerformed = filterContext.ActionDescriptor.RouteValues["action"];
 Debug.WriteLine(actionPerformed + " started");
 }

 public override void OnActionExecuted(ActionExecutedContext filterContext)
 {
 string actionPerformed = filterContext.ActionDescriptor.RouteValues["action"];
 Debug.WriteLine(actionPerformed + " finished");
 }
}
```

When you navigate to the **Index** action in the **MyController** controller, the **Index started** and **Index finished** lines are displayed in the Visual Studio Output Window.

You can use the **ResultExecutedContext** parameter to get the content that will be returned to the browser.

The following code shows how the content returned to the browser can be retrieved in the **OnResultExecuted** event handler:

### A simple custom action filter

```
public class SimpleActionFilter : ActionFilterAttribute
{
 public override void OnResultExecuted(ResultExecutedContext filterContext)
 {
 ContentResult result = (ContentResult)filterContext.Result;
 Debug.WriteLine("Result: " + result.Content);
 }
}
```

When you navigate to the **Index** action in the **MyController** controller, the **Result: some text** text is displayed in the Visual Studio Output Window.

Writing to the Visual Studio Output Window might be insufficient in many cases. Writing the output to other targets, for example a log file, may be a better choice in most situations.

The following code shows how you can change the content of the **SimpleActionFilter** class so that it writes the output to a log file that is located at **c:\logs\log.txt**:

### Writing to a log file

```
public class SimpleActionFilter : ActionFilterAttribute
{
 public override void OnActionExecuting(ActionExecutingContext filterContext)
 {
 string actionPerformed = filterContext.ActionDescriptor.RouteValues["action"];

 using (FileStream fs = new FileStream("c:\\\\logs\\\\log.txt", FileMode.Create))
 {
 using (StreamWriter sw = new StreamWriter(fs))
 {
 sw.WriteLine(actionPerformed + " started");
 }
 }
 }

 public override void OnActionExecuted(ActionExecutedContext filterContext)
 {
 string actionPerformed = filterContext.ActionDescriptor.RouteValues["action"];

 using (FileStream fs = new FileStream("c:\\\\logs\\\\log.txt", FileMode.Append))
 {
 using (StreamWriter sw = new StreamWriter(fs))
 {
 sw.WriteLine(actionPerformed + " finished");
 }
 }
 }

 public override void OnResultExecuting(ResultExecutingContext filterContext)
 {
 using (FileStream fs = new FileStream("c:\\\\logs\\\\log.txt", FileMode.Append))
 {
 using (StreamWriter sw = new StreamWriter(fs))
 {
 sw.WriteLine("OnResultExecuting");
 }
 }
 }

 public override void OnResultExecuted(ResultExecutedContext filterContext)
 {
```

```

ContentResult result = (ContentResult)filterContext.Result;
using (FileStream fs = new FileStream("c:\\logs\\log.txt", FileMode.Append))
{
 using (StreamWriter sw = new StreamWriter(fs))
 {
 sw.WriteLine("Result: " + result.Content);
 }
}
}

```

When you navigate to the **Index** action in the **MyController** controller, a file named **log.txt** will be created.

The following code shows the content of the **c:\logs\log.txt** log file:

#### Content of the log file

```

Index started
Index finished
OnResultExecuting
Result: some text

```

#### Using Dependency Injection in Filters

Until now we introduced one way to add filters: by instance. Such filters cannot get dependencies using their constructor provided by dependency injection.

 **Note:** Dependency injection was introduced in Module 3, "Configuring Middleware and Services in ASP.NET Core".

In case you need your filters to get dependencies using their constructor, you can add filters by type. To add a filter by type, you can use the **ServiceFilter** attribute.

For example, assume your filter needs to receive in its constructor an instance of **IHostingEnvironment**:

#### Using dependency injection in a filter

```

public class LogActionFilter : ActionFilterAttribute
{
 private IHostingEnvironment _environment;

 public LogActionFilter(IHostingEnvironment environment)
 {

 }
}

```

To apply the **LogActionFilter** filter to an action named Index, you can use the **ServiceFilter** attribute:

#### Applying the LogActionFilter filter to an action

```

public class CityController : Controller
{
 [ServiceFilter(typeof(LogActionFilter))]
 public IActionResult Index()
 {
 }
}

```

## Demonstration: How to Create and Use Action Filters

In this demonstration, you will learn how to create a new action filter and use it on an action.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Create and Use Action Filters” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD04\\_DEMO.md#demonstration-how-to-create-and-use-action-filters](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD04_DEMO.md#demonstration-how-to-create-and-use-action-filters).

# Lab: Developing Controllers

## Scenario

You have been asked to add controllers to a new application. The controllers should include actions that return a view, also add an action that returns the photo as a .jpg file to show on a webpage, and an action that redirects to another action in another controller. Additionally, you have been asked to configure routes in a variety of ways.

The members of your development team are new to ASP.NET Core 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 an external file whenever an action runs. To achieve this, you will add an action filter.

## Objectives

After completing this lab, you will be able to:

- Add an MVC controller to a web application
- Write actions in an MVC controller that respond to user operations
- Add custom routes to a web application
- Write action filters that run code for multiple actions

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD04\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD04_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD04\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD04_LAK.md).

## Exercise 1: Adding Controllers and Actions to an MVC Application

### Scenario

In this exercise, you will create the MVC controller that handles user operations. You will also add the following actions:

- **Index.** This action displays the **Index** view.
- **Display.** This action takes an ID to find a single **City** object. It passes the **City** object to the **Display** view.
- **GetImage.** This action returns the photo image from the service as a .jpg file.

The main tasks for this exercise are as follows:

1. Add controllers to an MVC application.
2. Add actions to a controller.
3. Change actions to get a parameter.
4. Change an action to redirect to another action in another controller.
5. Use a service.
6. Store the result in a **ViewBag** property.
7. Run the application.

## Exercise 2: Configuring Routes by Using the Routing Table

### Scenario

An important design priority for the application is that the visitors should be able to easily and logically locate cities. To implement these priorities, you have been asked to configure routes by using the routing table that enables the entry of user-friendly URLs to access cities.

The main tasks for this exercise are as follows:

1. Add a controller with an action.
2. Run the application.
3. Register new routes in the routing table.
4. Run the application and verify that the new route works.

## Exercise 3: Configuring Routes by Using Attributes

### Scenario

In addition to configuring routes by using the routing table, you have been asked to configure routes by using attributes as well to enable the entry of user-friendly URLs.

The main tasks for this exercise are as follows:

1. Apply custom routes to a controller by using attributes.
2. Run the application and verify that the new routes work.

## Exercise 4: Adding an Action Filter

### Scenario

Your development team is new to ASP.NET Core 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 and action names in an external file to help with this problem. In this exercise, you will create an action filter for this purpose.

The main tasks for this exercise are as follows:

1. Add an action filter class.
2. Add a handler for the **OnActionExecuting** event.
3. Add a handler for the **OnActionExecuted** event.
4. Add a handler for the **OnResultExecuted** event.
5. Apply the action filter to the controller action.
6. Run the application and verify that the new filter works.

**Question:** You decided to add a new action to the **CityController** controller. The action gets two parameters of type int. Are the routes that are currently configured in the application sufficient to handle the requests to this action or will you need to configure a new route?

**Question:** A new controller with an action is added to the **WorldJourney** MVC application. You want to ensure that every access to the new action is written to the external file. How can you achieve this?

## Module Review and Takeaways

In this module, you learned about the pivotal role that controllers, actions and routing play in the construction of an MVC web application. Routing determines the action that handles a request. 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 rendering. In the next module, you will learn how to create and code views to implement the user interface for your application.

### Review Question

**Question:** An MVC application contains many actions that have no parameters. You want to set a routing rule to match these actions. What should you do?

### Best Practice

The Visual Studio project templates include a folder named Controllers. Programmers should create their controllers in this folder or its subfolders. MVC relies on convention over configuration. Therefore, we recommend adhering to the conventions. Visual Studio places controllers in the **ProjectName.Controllers** namespace, by default.

### Common Issues and Troubleshooting Tips

| Common Issue                                                             | Troubleshooting Tip                                                                                                                                                                   |
|--------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| You navigate to an existing action, but get an HTTP 404 Not Found error. | Verify that the routes are configured correctly in the MVC application.                                                                                                               |
| A configuration-based route never takes effect.                          | You must ensure that you add configuration-based routes in the correct order. In general, the most specific routes should be added first and the least specific should be added last. |

# Module 5

## Developing Views

### Contents:

|                                              |       |
|----------------------------------------------|-------|
| Module Overview                              | 05-1  |
| Lesson 1: Creating Views with Razor Syntax   | 05-2  |
| Lesson 2: Using HTML Helpers and Tag Helpers | 05-12 |
| Lesson 3: Reusing Code in Views              | 05-20 |
| Lab: Developing Views                        | 05-29 |
| Module Review and Takeaways                  | 05-31 |

## Module Overview

Views are one of the three major components of the Model-View-Controller (MVC) programming model. You can define the user interface for your web application by creating views. A view is a combination of HTML markup and C# code that runs on a web server. Therefore, to create a view, you need to know how to write the HTML markup and C# code and use the various helper classes that are built into MVC. You also need to know how to create partial views and view components, which render sections of HTML that can be reused in your web application.

### Objectives

After completing this module, you will be able to:

- Create an MVC view and add Razor markup to it to display data to users.
- Use HTML helpers and tag helpers in a view.
- Reuse Razor markup in multiple locations throughout an application.

# Lesson 1

## Creating Views with Razor Syntax

When an MVC web application receives a request, a controller action processes the request. Often, the controller action passes a model object to a view. The view builds the data by inserting the properties from the model object and other sources into the HTML markup. Then, the view renders the completed data to the browser. The browser displays the data it receives from the view.

To create views, you must understand how MVC interprets the code you place in views and how a completed HTML page is built. By default, the Razor view engine performs this interpretation. The Razor view engine lets you control the rendered HTML.

### Lesson Objectives

After completing this lesson, you will be able to:

- Add views to an MVC web application.
- Use the @ symbol in a Razor view to treat code as content.
- Describe the features of the Razor syntax.
- Inject services to views.

### Adding Views

In an MVC application, controllers handle all the application logic. Each controller contains one or more action methods, and an action handles the incoming web requests. Views handle the presentation logic and should not contain any application logic. Views are used to render a response to the browser.

 **Note:** Controllers and actions are covered in Module 4, "Developing Controllers".

- Views handle the presentation logic
- View files have a .cshtml extension
- Within a controller, you can use the **View** method

```
public class ProductController : Controller
{
 public IActionResult Index()
 {
 return View();
 }
}
```

In an MVC application, there is usually one controller for every model class. For example, a model class named **Product** usually has a controller called **ProductController** that contains all the actions relevant to products. There might be some controllers that do not correspond to any model classes. However, each controller can have multiple views. For example, you might want to create the following views for **Product** objects:

- **Details.** The **Details** view can display a product, its price, catalog number, and other details.
- **Create.** The **Create** view can enable users to add a new product to the catalog.
- **Edit.** The **Edit** view can enable users to modify the properties of an existing product.
- **Delete.** The **Delete** view can enable users to remove a product from the catalog.
- **Index.** The **Index** view can display all the product objects in the catalog.

 **Note:** Models will be covered in Module 6, "Developing Models".

By convention, all the views in an MVC application reside within the top-level **Views** folder. Within this folder, there is a folder for each controller that is used in the application. In the preceding example, the **Views** folder would contain a **Product** folder and this folder would contain the **Details**, **Create**, **Edit**, **Delete**, and **Index** views.

If you use the Razor view engine and the C# language, views files are created with a .cshtml extension. Other view engines might use different extensions.

## How to Create a View File

You can create a view file in Microsoft Visual Studio by performing the following steps:

1. In Solution Explorer, select the folder where you want to create the view. For example, for a controller named **ProductController**, MVC expects view files to be located in the **Views/Product** folder. If such a folder does not exist yet, create the folder.
2. Right-click the selected folder, point to **Add**, and then select **View**.

Alternatively, you can create a view file for a particular controller action by opening the controller code file, right-clicking the action, and then clicking **Add View**. If you create a view by using the controller code file, some properties in the **Add View** dialog box are filled in by default.

The following table describes the properties that you must complete in the **Add View** dialog box:

| Property                   | Description                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| View name                  | This is the name of the view. The view file name is this name with the appropriate extension added. The name you choose should match the name returned by the corresponding controller action. If the action controller does not specify the name of the view to use, MVC assumes the name of the view matches the name of the controller action.                                                   |
| Model class                | If you create a strongly-typed view, you need to specify the model class to bind to the view. Visual Studio will use this class when it formulates IntelliSense prompts and checks for compile-time errors.                                                                                                                                                                                         |
| Template                   | A template is a basic view that Visual Studio can use to create the view. If you specify a model class for the view, Visual Studio can create simple templates for <b>Create</b> , <b>Edit</b> , <b>Details</b> , <b>Delete</b> , and <b>List</b> views. If you don't specify the model class for the view, you may choose to build a view from the <b>Empty (without model)</b> scaffold template. |
| Reference script libraries | When you select this check box in MVC 5, references to common client-side script files are included in the project. In ASP.NET Core MVC, this check box has no effect.                                                                                                                                                                                                                              |
| Create as a partial view   | A partial view is a section of Razor code that you can re-use in multiple views in your application.                                                                                                                                                                                                                                                                                                |
| Use a layout page          | A layout page can be used to impose a standard layout and branding on many pages within the web application.                                                                                                                                                                                                                                                                                        |

Within a controller, you can use the **View** method to create a **ViewResult** object. The view renders data to the browser.

The following code shows a simple controller class that calls the **View** method to create a **ViewResult** object:

### Using the View method

```
public class ProductController : Controller
{
 [Route("Product/Index")]
 public IActionResult Index()
 {
 return View();
 }
}
```



**Note:** Routing is covered in Module 4, "Developing Controller".

The following code shows the content of a file named **Index.cshtml** that is located in the **Views/Product** folder:

### A view

```
<!DOCTYPE html>

<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Index</title>
</head>
<body>
 Hello from the Index view
</body>
</html>
```

If a user requests the relative URL, /Product/Index, the text, "Hello from the Index view," will be sent to the browser.

## Differentiating Server-Side Code from HTML

The Razor view engine interprets view files and runs any server-side code contained in the view files. To do this, the Razor view engine must distinguish the server-side code from the HTML content. The HTML content should be sent to the browser unchanged. The Razor view engine looks for the @ symbol to identify the server-side code.

In the following code example, the Razor view engine runs the line that has the @ symbol as C# code on the web server. The text "Result: ab" is sent to the browser:

- Razor identifies server-side code by looking for the @ symbol
- Razor distinguishes the server-side code from the HTML content that is sent to the browser unchanged

```
<body>
 @for (int i = 0; i < 5; i++)
 {
 @i
 }
</body>
```

### Using @ to declare server-side code

```
Result: @String.Concat("a", "b")
```

A section of code marked with the @ symbol is referred to as a Razor code expression. In the Razor syntax, you mark the start of a Razor code expression with the @ symbol. However, there is no expression ending symbol. Instead, Razor infers the end of a code expression by using a fairly sophisticated parsing engine.

In the following code example, the text "Result: ab Razor knows that I'm an html!!! ab I'm a regular text – Razor is smart enough to know I'm a server-side code. Ab" is sent to the browser:

### Using Razor

```
Result: @String.Concat("a", "b") Razor knows that I'm an html!!!
@String.Concat("a", "b") I'm a regular text - Razor is smart enough to know I'm not a
server-side code. @String.Concat("a", "b")
```

### Writing a for Loop by Using Razor

You can write for loops by using the Razor syntax if you know exactly how many times you want to loop. The for loop syntax in Razor is very simple.

The following code shows how you can write a for loop by using the Razor syntax. The text "01234" is sent to the browser:

### For Loop Razor syntax

```
@for (int i = 0; i < 5; i++)
{
 @i
}
```

In the above example, the Razor engine translates the @i expression to the value of i variable. The value of i is sent to the browser. You can even use expressions such as @i\*2, which multiplies the value of i with 2.

Razor has sophisticated logic to distinguish code from content and often the @ symbol is all that is required. However, occasionally, you might find that an error occurs because Razor misinterprets content as code. To fix such errors, you can use the @: delimiter, which explicitly declares a line as content and not as code.

The following code shows how you can use the @: delimiter. The text "iiiii" is sent to the browser:

### Using the @: delimiter

```
@for (int i = 0; i < 5; i++)
{
 @:i
}
```

Razor easily differentiates content from code. For example, even within a Razor code block, when you add an HTML element such as <span>, Razor interprets the text as content. Usually, you do not have to use the @: delimiter to make this explicit.

In the following code the text "iiii i" is sent to the browser:

### A variable inside an HTML element not prefixed by the @ symbol

```
@for (int i = 0; i < 5; i++)
{
 i
}
```

If you want to send the value of the *i* variable to the browser, you can place an @ symbol before *i*.

In the following code, the text "0 1 2 3 4" is sent to the browser:

### A Variable inside an HTML Element Prefixed by the @ Symbol

```
@for (int i = 0; i < 5; i++)
{
 @i
}
```

### Modifying the Interpretation of Code and Content

Sometimes, you might need to modify the logic that Razor uses to interpret code expressions. For example, if you want to display an @ symbol in the browser, you use @@. Razor interprets this as an escape sequence and renders a single @ symbol. This technique is useful for rendering email addresses.

The following code shows how you can use @@ to render a single @ symbol. The text "@ @ @ @ @" is sent to the browser:

### Use @@ in a view

```
@for (int i = 0; i < 5; i++)
{
 @@
}
```

If you want to declare several lines as content, use the <text> tag instead of the @: delimiter. Razor removes the <text> and </text> tags before returning the content to the browser.

The following code shows how you can use the <text> tag. The text "i i i i" is sent to the browser:

### Using the <text> tag

```
@for (int i = 0; i < 2; i++)
{
 <text>
 i
 i
 </text>
}
```

## Features of the Razor Syntax

Razor includes many useful features that you can use to control the way in which ASP.NET Core MVC renders your view as HTML. These features include the following:

- Razor comments
- Implicit code expressions and parentheses
- Razor code blocks and conditions
- Razor loops

A sample code block displaying the features of Razor

```
/* Some more Razor examples */

 Price including Sale Tax: @ViewBag.Price * 1.2

 Price including Sale Tax: @(ViewBag.Price * 1.2)

 @{
 int i = 5;
 int j = 6;
 int z = i + j;
 @z
}
```

## Razor Comments

You might want to include comments in your Razor code to describe it to other developers in your team. This is an important technique that improves developer productivity by making the code easy to understand.

You can declare a Razor comment by using the @\* delimiter, as shown in the following code example:

### A Razor comment

```
@* This text will not be rendered by the Razor view engine because this is a comment. *@
```

## Implicit Code Expressions and Parentheses

Razor uses implicit code expressions to determine parts of a line that are server-side code. Usually, implicit code expressions render the HTML you want, but occasionally, you might find that Razor interprets an expression as HTML when it should be run as server-side code.

For example, consider the following controller class that sets the value of the ViewBag.Price property to 2:

### A Controller class

```
public class ProductController : Controller
{
 [Route("Product/Index")]
 public IActionResult Index()
 {
 ViewBag.Price = 2;
 return View();
 }
}
```

In the following code, the Razor view engine renders the text "Price Including Sales Tax: 2 \* 1.2" to the browser:

### An implicit code expression

```
Price Including Sales Tax: @ViewBag.Price * 1.2
```

You can control and alter this behavior by using parentheses to enclose the expression so that Razor can evaluate the expression.

For example, in the following code, the Razor view engine renders the text "Price Including Sales Tax: 2.4" to the browser:

### Using parentheses to explicitly delimit expressions

```
Price Including Sales Tax: @(ViewBag.Price * 1.2)
```

## Razor Code Blocks and Conditions

If you want to write multiple lines of server-side code without prefacing every line with the @ symbol, you can use a Razor code block.

For example, in the following code, the Razor view engine sends the text "11" to the browser:

### A Razor code block

```
@{
 // Razor interprets all text within these curly braces as server-side code
 int i = 5;
 int j = 6;
 int z = i + j;
 @z
}
```

```
}
```

Razor includes code blocks that run conditional statements.

For example, in the following code, the Razor view engine sends the text "5" to the browser:

### A Razor If code block

```
@{
 int i = 5;
 int j = 6;

 @if (i < j)
 {
 @i
 }
}
```

### Razor Loops

Razor also includes code blocks that loop through collections. You can loop through all the objects in an enumerable collection by using the **foreach** code block.

For example, consider the following controller class that sets the value of the **ViewBag.Names** property to contain a list of strings:

### A Controller class

```
public class NameController : Controller
{
 [Route("Name/Index")]
 public IActionResult Index()
 {
 ViewBag.Names = new List<string>() { "name1", "name2", "name3" };
 return View();
 }
}
```

You can loop through all the strings in the **ViewBag.Names** property by using the **foreach** code block, as shown in the following code. The text "name1 name2 name3" is sent to the browser:

### A Razor Foreach code block

```
@foreach (string name in ViewBag.Names)
{
 @name
}
```

Razor loops are useful for creating index views, which display many objects of a particular model class. A product catalog page, which displays many products from the database, is a good example of an index view. To implement an index view, a controller action passes an enumerable collection of objects to the view.

For example, consider the following model class:

### A Model class

```
namespace Models
{
 public class Product
 {
 public string Name { get; set; }
 // Other fields are omitted
 }
}
```

```
}
```

The following controller class creates a list of products and passes it to a view:

### A Controller class

```
public class ProductController : Controller
{
 [Route("Product/Index")]
 public IActionResult Index()
 {
 Product p1 = new Product() { Name = "Product1" };
 Product p2 = new Product() { Name = "Product2" };
 List<Product> products = new List<Product>() { p1, p2 };
 return View(products);
 }
}
```

The following view gets the products from the controller. The Razor view engine sends the text "product1 product2" to the browser:

### Iterating a Model by using Razor

```
@model IEnumerable<Models.Product>

@foreach (var product in Model)
{
 @product.Name
}
```

## Demonstration: How to Use the Razor Syntax

In this demonstration, you will learn how to write views by using the Razor syntax.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use the Razor Syntax" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_DEMO.md#demonstration-how-to-use-the-razor-syntax](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD05_DEMO.md#demonstration-how-to-use-the-razor-syntax).

## Alternative View Engines

A view engine is a component of the MVC framework that is responsible for locating view files, running the server-side code they contain, and rendering HTML that the browser can display to the user. The Razor view engine is the default view engine in MVC and is highly flexible and efficient. Furthermore, it is easy to learn if someone has experience in HTML and C#. Also, Visual Studio provides good IntelliSense feedback for Razor code.

- A view engine is a component of the MVC framework that is responsible for:
  - Locating view files
  - Running the server-side code that view files contain
  - Rendering HTML that the browser can display to the user
- Razor is the default view engine in MVC
- It is possible to create a custom view engine
  - Need to implement the **IViewEngine** interface

Despite its many advantages, some developers might prefer to use a different view engine. The reasons can include:

- View Location Logic. Razor assumes that all views are located in the **Views** top-level folder. Within this, Razor assumes that each controller has a separate folder. For example, views for **ProductController** are located in the **Product** folder within **Views**. Razor also looks for some views in the **Shared** subfolder. These views can be used by more than one controller. If you do not want to use this view location logic, you can use an alternate view engine.
- View Syntax. Each view engine uses a different syntax in view files. For example, in Razor the @ symbol delimits the server-side code. Other view engines can use different delimiters. You might prefer one syntax over another based on the technologies that you or your team previously worked with.

You can create your own view engine to implement custom logic for locating views or to implement custom syntax for view files. To create a custom view engine, you should create a class that implements the **IViewEngine** interface. The **IViewEngine** interface defines the contract for a view engine. It contains a **FindView** method, which finds a specified view by using a specified action context. The **FindView** method returns a **ViewEngineResult** object.

Custom view engines are rarely created because they require the developers to write sophisticated string-parsing code. The default view engine, Razor, is powerful and flexible. The default logic for finding view files is also simple and rarely needs modification.

## Dependency Injection into Views

With Razor views, you can use dependency injection to inject services directly into views. By injecting a service directly into a view, you can call it whenever you wish by using the Razor logic, potentially gaining immediate access to the functionality of the service, without needing to rely on a controller.

A very common use case where this is useful is localization services. As part of localization requirements, you will often want to localize large sections of text in a view, and it will be very awkward to have the model manage the text for specific segments. By using a localization service in a view, you can easily maintain localization in the view, without having to create complex controller logic.

On the opposite end of the spectrum, you should also avoid using services on a view with the intention to present data. By using services to retrieve data, you are ignoring the principles of MVC and creating a direct dependency between our view and the data.

Mostly, you will have to use only stateless services in views, and mainly services designed to perform transformations rather than create data. For instance, a service that performs formatting on dates for an application is logical to keep in views because this kind of service will be repeated often, and result in a lot of view-related logic ending in a controller. On the other hand, a service that retrieves data from a database is a bad choice to use in views because retrieving data off a database is a costly operation. If this operation is repeated multiple times, the performance of a view will suffer.

Dependency injection into views is performed by using *directives*, which are special keywords that you can use after the @ symbol to perform specific actions. To inject services, you will use the **inject** directive.

- ASP.NET Core supports dependency injection into views
- You can inject a service into a view using the **@inject** directive
  - `@inject <type> <instance name>`

Here is the format for the **inject** directive:

```
@inject <type> <instance name>
```

The type is the interface of the service that you wish to inject. To find a type, you will have to use the full path for the namespace of the interface.

The instance name is an alias you create to refer to a service. Throughout a view, calling the alias will grant you access to the methods exposed by an interface.

After you inject an interface, you can access the service at any time inside the code contained by the @ symbol.

The following code is an example of a service that formats numbers to be comma-separated:

### Formatting service

```
namespace Services
{
 public interface IFormatNumber
 {
 string GetFormattedNumber(int number);
 }
 public class FormatNumber : IFormatNumber
 {
 public string GetFormattedNumber(int number)
 {
 return string.Format("{0:n0}", number);
 }
 }
}
```

The following code is an example of a view that calls the **FormatNumber** service:

### View dependency injection

```
@inject Services.IFormatNumber formatNumber

@formatNumber.GetFormattedNumber(1234535334)
```

This example presents a simple service for formatting numbers as string. Specifically, it will add commas to numbers every third digit. In the second snippet, the service is injected into a view. If this view is called, the string "1,234,535,334" will be displayed in the browser. In particular, take note that the service namespace, **Services**, matches up with the namespace in the **@inject** directive.

 **Note:** As an additional note, you can also add a **using** directive with the @ symbol. This can allow you to specifically apply a namespace to the view, just as you would do in a C# class. For example, in this example, if you used **@using Services**, you would be able to update the **@inject** directive to **@inject IFormatNumber formatNumber**.

 **Best Practice:** Generally, you will want injected services to be placed at the top of the HTML. This is mainly to improve the view legibility. While the Razor engine can handle injection at the end of a page, using it can lead to unclear code.

## Lesson 2

# Using HTML Helpers and Tag Helpers

ASP.NET Core MVC includes many tag helpers and HTML helpers that you can use in views. Helpers include common logic that makes it easier for you to render HTML in MVC applications. In this lesson, some tag helpers and HTML helpers will be introduced.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use HTML helpers in a view.
- Use tag helpers in a view.
- Use the **Html.ActionLink** and **Url.Action** helpers to call a controller action.

### Introduction to HTML Helpers and Tag Helpers

In general, views are composed of correctly formed HTML elements. However, because the views are not static (like normal HTML) and can even run code at creation time, you might sometimes need HTML behaviors. But creating the entire behavior manually could be time consuming. Examples of such events include creating links, managing forms and controls inside the forms and even the forms themselves. Normally, these may require you to perform some extra maintenance. For example, the paths in links may change and forms require JavaScript to correctly manage.

- HTML helpers:
  - Use a Razor syntax
  - Make it easier to identify areas of code
  - Does not require explicit enabling of the feature

- Tag helpers:
  - Use an HTML-like syntax, as well as tag properties
  - Require explicit usage of a directive
  - Create more easily legible HTML with less immediately apparent code

To resolve this issue, and allow you to develop applications faster with less repetition on repeated logic and less dependency on JavaScript, you can use HTML helpers and tag helpers.

HTML helpers are C# methods that you can use to create common and complex HTML elements, such as links and various input controls and forms, and manage the elements and their relationship to controls. This is done by using Razor syntax.

Tag helpers are an alternative to HTML helpers. Tag helpers help you add properties to existing HTML elements and custom html elements, instead of adding Razer code. This will help you keep your html more consistent. As a result, code with tag helpers looks a lot more like HTML code than HTML helpers. Regardless of this, the end result is consistent between the two. To use tag helpers, use the **@addTagHelper** directive, which is used to load tag helper classes.

## Using HTML Action Helpers

HTML helpers are simple methods that, in most cases, return a string. This string is a small section of HTML that the view engine inserts into the view file to generate a completed webpage. It is not mandatory to use HTML helpers and you have the flexibility to create views that render any type of HTML content without using any HTML helpers.

You can use the **Html.ActionLink** and **Url.Action** helpers to render HTML that calls controller actions.

### • **Html.ActionLink()**

```
@Html.ActionLink("Click here to view photo 1",
"Display", new { id = 1 })
```



```

Click here to view photo 1

```

### • **Url.Action()**

```

```



```

```

## The **Html.ActionLink** Helper

When a user requests an ASP.NET Core MVC web application, the ASP.NET Core MVC framework forwards the request to the appropriate controller and action, and passes the correct parameters to the action. Users can make such requests by entering a URL in the address bar of a web browser, but they might not know the name of the controller, action, or parameters. In such cases, users can click a link to make the request. You can use the **Html.ActionLink** helper to render a link to an action. The helper returns an **<a>** element with the correct *href* parameter value.

The following code example shows a controller class with two action methods:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 return View();
 }
 [Route("Home/AnotherAction")]
 public IActionResult AnotherAction()
 {
 return Content("AnotherAction result");
 }
}
```

In the following code example, the **Html.ActionLink** helper is used in the **Index** view to render an **<a>** element:

### Using the **Html.ActionLink** helper

```
@Html.ActionLink("Press me", "AnotherAction")
```

If a user requests the relative URL `/Home/Index`, the following HTML will be sent to the browser:

```
Press me.
```

If the user clicks the **Press me** link, the **AnotherAction** action of the **HomeController** controller will be called, and the text "AnotherAction result" will appear on the browser.

### Using **Html.ActionLink** to Call Actions in Other Controllers

In the previous example, the **@Html.ActionLink** helper is used to call an action in the same controller. You can also use the **@Html.ActionLink** helper to call actions in other controllers.

The following code example shows two controllers. Each controller contains one action. The action of the first controller returns a **ViewResult** object and the action of the second controller returns a **ContentResult** object:

### The Controllers code

```
public class FirstController : Controller
{
 [Route("First/Some")]
 public IActionResult Some()
 {
 return View();
 }
}

public class SecondController : Controller
{
 [Route("Second/Index")]
 public IActionResult MyAction()
 {
 return Content("Second Controller");
 }
}
```

The following code example shows how the **Html.ActionLink** helper is used in the **Some** view to render an **<a>** element. If the user clicks the link in the browser, MVC will direct the request to the **MyAction** action in the **SecondController** controller:

### Using the **HTML.ActionLink** helper to call an action in another controller

```
@Html.ActionLink("Press the link", "MyAction", "Second")
```

If a user requests the relative URL, /First/Some, the following HTML will be sent to the browser:

### HTML sent to browser

```
Press the link
```

If the user clicks the **Press the link** link, the text **Second Controller** appears in the browser.

### Passing Parameters to **Html.ActionLink**

You can pass parameters to the **Html.ActionLink** helper.

The following code example shows a controller class with two action methods. The second action method, **Display**, gets a parameter named *id* of type **int**:

### A Controller class

```
public class PhotoController : Controller
{
 [Route("Photo/Choose")]
 public IActionResult Choose()
 {
 return View();
 }

 [Route("Photo/Display/{id}")]
 public IActionResult Display(int id)
 {
 string res = $"Photo number {id} was chosen";
 return Content(res);
 }
}
```

In the following code example, the **Html.ActionLink** helper is used to render an **<a>** element. If the user clicks the link, the **Display** action in the **PhotoController** controller will be called, and an integer value will be passed to the *id* parameter:

### Using **Html.ActionLink** to pass a parameter

```
@Html.ActionLink("Click here to view photo 1", "Display", new { id = 1 })
@Html.ActionLink("Click here to view photo 2", "Display", new { id = 2 })
```

If a user requests the relative URL, /Photo/Choose, the following HTML will be sent to the browser:

### HTML sent to the browser

```
Click here to view photo 1
Click here to view photo 2
```

If the user clicks the **Click here to view photo 1** link, the text "Photo number 1 was chosen" appears in the browser. If the user clicks the **Click here to view photo 2** link, the text "Photo number 2 was chosen" appears in the browser.

### The **Url.Action** Helper

The code in a view can contain the **Url.Action** helper. You can use the **Url.Action** helper to render a URL without the enclosing **<a>** element. To render the URL, the routing engine is used.

The following code shows how the **Url.Action** helper can be used in a view:

### Using the **Url.Action** helper

```
@Url.Action("actionName", "controllerName")
```

You can use the **Url.Action** helper to populate the **src** attribute of an **<img>** element or a **<script>** element. You also can use this helper whenever you want to formulate a URL to a controller action without rendering a hyperlink.

The following code example shows how to use the **Url.Action** helper within the **src** attribute of an **img** element:

### Using the **Url.Action** helper in the **src** attribute

```

```

The following code example shows the **GetImage** action:

### The **GetImage** action

```
public IActionResult GetImage(int id)
{
 string path = GetPath(id);
 return File(path, "image/jpeg");
}
```

## Demonstration: How to Use HTML Helpers

In this demonstration, you will learn how to use the **Html.ActionLink** helper in a view, and how to use the **Url.Action** helper to set the **src** attribute of an image.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use HTML Helpers" on the following page:  
[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_DEMO.md#demonstration-how-to-use-html-helpers](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD05_DEMO.md#demonstration-how-to-use-html-helpers).

## Using Tag Helpers

Views are used to render data to a browser.

Though the data that is rendered to the browser is invariably HTML, views usually contain a mixture of server-side code and HTML. As you learned earlier, it is possible to embed server-side code in a view by using the Razor syntax. You also learned that HTML helpers are server-side extension methods that can be placed inside a view. However, to understand HTML helpers, it is important to be familiar with a programming language such as C#.

An alternative to HTML helpers are tag helpers.

The role of tag helpers is similar to the role of HTML helpers, which is to embed server-side code inside a view. However, tag helpers look similar to regular HTML elements. Though they look like HTML elements, tag helpers are implemented by using programming languages such as C#. For example, the HTML produced by the HTML helper **@Html.ActionLink("Press me", "AnotherAction")** and the tag helper **<a asp-action="AnotherAction">Press me</a>** are identical.

ASP.NET Core MVC includes several predefined tag helpers. These predefined tag helpers are located in the **Microsoft.AspNetCore.Mvc.TagHelpers** namespace. To use them, you need to add the **@addTagHelper** directive to the view to define the tag helpers that this view will use. Alternatively, you can add the **@addTagHelper** directive to a **\_ViewImports.cshtml** file that is located under the **Views** folder, so that it is available to all the views.

One of the predefined tag helpers is **AnchorTagHelper**. This tag helper is associated with the HTML **a** element. In the example, **<a asp-action="AnotherAction">Press me</a>**, the **AnchorTagHelper** tag helper is used. **AnchorTagHelper** is essentially a C# class that contains several attributes such as **asp-action**, which indicates the name of the action, and **asp-controller**, which indicates the name of the controller.

The following example shows how to use a tag helper and an HTML helper such that the same HTML will be rendered to the browser.

- Tag helpers are an alternative to HTML helpers
- Tag helpers look like regular HTML elements
- The following HTML helper and tag helper produce the same HTML:

HTML helper:

```
@Html.ActionLink("Press me", "AnotherAction")
```

Tag helper:

```
<a asp-action="AnotherAction">Press me
```

The following code example shows a controller class with two action methods:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 return View();
 }
 [Route("Home/AnotherAction")]
 public IActionResult AnotherAction()
 {
 return Content("AnotherAction result");
 }
}
```

The following code example shows the **Html.ActionLink** HTML helper followed by a tag helper:

### Using a Tag Helper in a view

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@Html.ActionLink("Press me", "AnotherAction")
<a asp-action="AnotherAction">Press me
```

If a user requests the relative URL /Home/Index, the following HTML will be sent to the browser:

### HTML sent to browser

```
Press me
Press me
```

Notice that the HTML produced by the HTML helper and the tag helper are identical.

### Calling Actions in Other Controllers

In the example above, you saw how you can use the HTML helper and the tag helper to call an action in the same controller. You can use both of them to call actions in other controllers. The HTML produced for both helpers is identical. The following example shows how to use a HTML helper and a tag helper to call actions in other controllers.

The following code example shows two controllers. Each controller contains one action. The action of the first controller returns a **ViewResult** object and the action of the second controller returns a **ContentResult** object:

### The Controllers code

```
public class FirstController : Controller
{
 [Route("First/Some")]
 public IActionResult Some()
 {
 return View();
 }
}

public class SecondController : Controller
{
 [Route("Second/Index")]
 public IActionResult MyAction()
 {
 return Content("Second Controller");
 }
}
```

```
}
```

The following code example shows the **Html.ActionLink** HTML helper followed by a tag helper:

### Using a HTML Helper and a Tag Helper to call actions in other controllers

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@Html.ActionLink("Press the link", "MyAction", "Second")
<a asp-action="MyAction" asp-controller="Second">Press the link
```

If a user requests the relative URL, /First/Some, the following HTML will be sent to the browser:

#### HTML sent to browser

```
Press the link
Press the link
```

Notice that the HTML produced by the HTML helper and the tag helper are identical.

### Passing Parameters to an Action

In the examples above, you saw how you can use the **AnchorTagHelper** tag helper to call an action. In case you want to pass parameters to an action, you can use the **asp-route-{value}** attribute. The **{value}** placeholder should be identical to the name of the route parameter. The following example shows how to pass a parameter to an action by using the **AnchorTagHelper** tag helper.

The following code example shows a controller class with two action methods. The second action method, **Display**, gets a parameter named *id* of type **int**:

#### A Controller class

```
public class PhotoController : Controller
{
 [Route("Photo/Choose")]
 public IActionResult Choose()
 {
 return View();
 }

 [Route("Photo/Display/{id}")]
 public IActionResult Display(int id)
 {
 string res = $"Photo number {id} was chosen";
 return Content(res);
 }
}
```

In the following code example, the **AnchorTagHelper** tag helper is used to with an **asp-route-{id}** attribute. If the user clicks the link, the **Display** action in the **PhotoController** controller will be called, and an integer value will be passed to the *id* parameter:

### Passing a parameter by using the AnchorTagHelper Tag Helper

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
<a asp-action="Display" asp-route-id="1">Click here to view photo 1
<a asp-action="Display" asp-route-id="2">Click here to view photo 2
```

If a user requests the relative URL /Photo/Choose, the following HTML will be sent to the browser:

#### HTML sent to the browser

```
Click here to view photo 1
Click here to view photo 2
```

If the user clicks the **Click here to view photo 1** link, the text "Photo number 1 was chosen" appears in the browser. If the user clicks the **Click here to view photo 2** link, the text "Photo number 2 was chosen" appears in the browser.

#### Using the \_ViewImports.cshtml File

If you add the **@addTagHelper** directive to a view, the tag helpers are available only in this view. In case you want that, the tag helpers will be available for all views in the **Views** folder and its sub directories. You can add the **@addTagHelper** directive to a **\_ViewImports.cshtml** file. The **\_ViewImports.cshtml** file should be located in the **Views** folder.

For example, instead of adding the **@addTagHelper** directive to the **Index.cshtml** file:

#### The @addTagHelper directive in the Index.cshtml file

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
<a asp-action="AnotherAction">Press me
```

You can add the **@addTagHelper** directive to the **\_ViewImports.cshtml** file in the **Views** folder:

#### The Views/\_ViewImports.cshtml file

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

In this case, you don't need any more to add the **@addTagHelper** directive to the **Index.cshtml** file:

#### **@addTagHelper directive is removed from the Index.cshtml file**

```
<a asp-action="AnotherAction">Press me
```

**AnchorTagHelper** is one of the built-in tag helpers. There are other built-in tag helpers such as **InputTagHelper**, which is associated with the **input** HTML element, and **ValidationMessageTagHelper**, which is used to display validation error messages. You also can create custom tag helpers. All tag helpers implement the **ITagHelper** interface, and many tag helpers inherit from the **TagHelper** class.

 **Note:** Other tag helpers, such as **InputTagHelper** and **ValidationMessageTagHelper**, will be covered in Module 6, "Developing Models".

## Demonstration: How to Use Tag Helpers

In this demonstration, you will learn how to add a tag helper to a view.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use Tag Helpers" on the following page:

[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_DEMO.md#demonstration-how-to-use-tag-helpers](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD05_DEMO.md#demonstration-how-to-use-tag-helpers).

## Lesson 3

# Reusing Code in Views

In a web application, you frequently render similar blocks of HTML content in different webpages. For example, in an e-commerce web application, you might want to display the most popular products at multiple locations such as the home page, the front page of the product catalog, and the top of the product search page. To render such HTML content in an MVC application, you can copy and paste code from one Razor view to other Razor views.

A better practice is to use a partial view. A partial view renders only a section of HTML content, which you can then insert into several other views at run time. Because a partial view is a single file that is reused in several locations in a web application, if you implement a change in one location, the change is updated in other locations. Partial views increase the manageability of MVC web applications and facilitate a consistent presentation of content throughout a web application.

A view component is another way to reuse code in views. View components are similar to partial views. However, view components have many benefits when compared to partial views. They can be treated as mini controllers because they render a chunk of data instead of rendering the whole response. A view component consists of a class and a view. The class is usually inherited from the **ViewComponent** class and the view uses Razor to call the methods in the class.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe how partial views enable you to reuse Razor code.
- Create partial views in an MVC web application.
- Use partial views in an MVC web application.
- Create a view component in an MVC web application.
- Use a view component in an MVC application.

### Creating Partial Views

You can use partial views to render identical or similar HTML content in different locations of your web application. Consider a web application that hosts articles and where users have the provision to comment on the articles. On each article's page, you want to display the comments for that specific article at the end of the page. Also, you want to display the most popular comments on any article on the home page. To address these needs, you can build a single comments partial view as follows:

- Use partial views to render identical HTML content in different locations of your web application
- Often created inside the **/Views/Shared** folder
- By convention, the names of partial views are prefixed with an underscore
  - For example: \_MyPartialView.cshtml

```
<p>In partial view</p>
```

1. First, create a partial view that displays a collection of comments in the required format.
2. Next, create a controller action that retrieves the most popular comments, and then passes them to the partial view.
3. Then, create another controller action that uses the article ID to find comments that are specific to that article. The controller action then passes this collection of comments to the partial view.

- Finally, call the appropriate actions from the home page and the article view.

Because the partial view renders a collection of comments, you can reuse it in various situations by passing different collections to it from the controller actions.

## Creating and Naming Partial Views

In Visual Studio, you can create a partial view by using the **Add View** dialog box, in the same way that you create any other view. By convention, the names of partial views are prefixed with an underscore, for example, \_CommentsList.cshtml. This convention is optional but is often helpful to distinguish partial views from regular views in Solution Explorer.

Partial views are often created inside the **/Views/Shared** folder in your site. Views in the **Shared** folder are accessible to many controllers, whereas views in the **/Views/Comment** folder are accessible only from the **CommentController**.

The following example shows a partial view that resides in the **\_MyPartialView.cshtml** file. The **\_MyPartialView.cshtml** file is located in the **/Views/Shared** folder:

### The **\_MyPartialView.cshtml** file

```
<p>In partial view</p>
```

## Strongly-typed and Dynamic Partial Views

Similar to other views, you can create strongly-typed partial views if you are sure that the views will always display the same model class. Visual Studio provides the most informative IntelliSense feedback and error-checking for strongly-typed partial views. A strongly-typed view has a declaration of the **@model** directive at the top of the file.

Alternatively, you can create a dynamic partial view if you are not sure that the partial view will always display the same model class. You can create dynamic partial views by omitting the **@model** directive declaration.

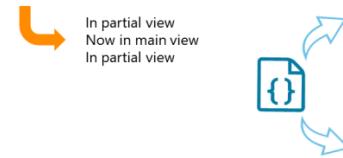
 **Note:** Models will be covered in Module 6, "Developing Models".

## Using Partial Views

You can use the **Html.PartialAsync** method to render a partial view within another view file. MVC inserts the HTML content that the partial view renders into the parent view and then returns the complete webpage to the browser. The **Html.PartialAsync** method is overloaded. The following code example demonstrates a version of the **Html.PartialAsync** method that gets the name of the partial view file, without its file extension, as a parameter.

You can use the **Html.PartialAsync()** method to render a partial view within another view file

```
@await Html.PartialAsync("_MyPartialView")
Now in main view
@await Html.PartialAsync("_MyPartialView")
```



The following code shows a simple controller, which calls a view that contains partial views:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 return View();
 }
}
```

The following lines of code show how to call the partial view that is located in the `_MyPartialView.cshtml` file:

### Using the `Html.PartialAsync` method

```
@await Html.PartialAsync("_MyPartialView")
Now in main view
@await Html.PartialAsync("_MyPartialView")
```

If a user runs the application and requests the relative URL, /Home/Index, the following text will display in the browser:

### Text displayed in the browser

```
In partial view
Now in main view
In partial view
```

### Passing Model Object to Partial Views

Models can be passed from action methods to views. A parent view will share the same model object with the partial view. This is a good way of sharing data between the controller action, view, and the partial view.

The following code shows a simple model class:

### A Model class

```
namespace Models
{
 public class SimpleModel
 {
 public int Value { get; set; }
 }
}
```

The following code shows a controller that creates an instance of a model and passes it to a view:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 SimpleModel model = new SimpleModel() { Value = 5 };
 return View(model);
 }
}
```

The following code shows how a model can be used in a view:

### Using the Model in the Index View

```
@model Models.SimpleModel

@await Html.PartialAsync("_MyPartialView")
Value received in index view from model is @Model.Value
@await Html.PartialAsync("_MyPartialView")
```

The following code shows how a model can be used in a partial view:

### Using the Model in the \_MyPartialView partial view

```
@model Models.SimpleModel

<p>Value received in partial view from model is @Model.Value</p>
```

If a user runs the application and requests the relative URL, /Home/Index, the following text will display in the browser:

### Text displayed in browser

```
Value received in partial view from model is 5
Value received in index view from model is 5
Value received in partial view from model is 5
```

### Using ViewBag in Partial Views

You can use the **ViewBag** and  **ViewData** collections to pass data between an action method and a view. A parent view will share the same **ViewBag** or  **ViewData** with the partial view. This is a good way to share data between a controller action, a view, and a partial view.

The following code shows a controller that passes data by using the **ViewBag** property:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 ViewBag.Message = "message";
 return View();
 }
}
```

The following code shows how a view can read the data that is passed to it by using the **ViewBag** property:

### Using the ViewBag property in the Index view

```
@await Html.PartialAsync("_MyPartialView")
Parent view received @ViewBag.Message
@await Html.PartialAsync("_MyPartialView")
```

The following code shows how a partial view can read the data that is passed to it by using the **ViewBag** property:

### Using the ViewBag property in the \_MyPartialView partial view

```
<p>Partial view received @ViewBag.Message</p>
```

If a user runs the application and requests the relative URL, /Home/Index, the following text will display in the browser:

#### Text displayed in browser

```
Partial view received message
Parent view received message
Partial view received message
```

## Demonstration: How to Create and Use Partial Views

In this demonstration, you will learn how to create and use partial views.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Create and Use Partial Views" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_DEMO.md#demonstration-how-to-create-and-use-partial-views](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD05_DEMO.md#demonstration-how-to-create-and-use-partial-views).

## Creating View Components

You can use view components to render identical or similar HTML content in different locations of your web application. In this manner, view components are similar to partial views. However, view components are much more powerful than partial views. A view component can be treated as a mini-controller which means that you can use it to render a small chunk of markup instead of a whole response.

A view component consists of two parts:

- A class. The class should be public and non-abstract. This class is usually derived from the **ViewComponent** base class. It can be annotated with the **ViewComponentAttribute** attribute. You can place the view component class in any folder of the project. However, a good practice is to place it in a folder named **ViewComponents**. The class should have a method called **InvokeAsync**, which defines its logic. The **ViewComponent** base class has a method named **View**, which returns a **ViewViewComponentResult** object. The **View** method can get a parameter that specifies the name of the partial view that needs to be rendered.
- A view. The view will typically be located in a folder under the **Views\Shared\Components** folder. This is because view components are usually not specific to a controller. The name of the folder should be the same as the name of the view component class. However, if the name of the class has the **ViewComponent** suffix, then the folder name should not include the **ViewComponent** suffix. For

- You can use view components to render identical or similar HTML content in different locations
- A view component consists of two parts:
  - A class
    - Should be public and non-abstract
    - Usually derived from the **ViewComponent** base class
    - Should have a method called **InvokeAsync**, which defines its logic
  - A view
    - Can be located in a folder under **Views\Shared\Components** folder
    - The name of the folder should be the same as the name of the view component class without the **ViewComponent** suffix

example, if the name of the view component class is **MyViewComponent**, the view can be in a folder named **Views\Shared\Components\My**.

The following code shows a view component class named **MyViewComponent**. The name of the partial view that will be rendered is **Default**:

### A ViewComponent class example

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ViewComponents
{
 public class MyViewComponent : ViewComponent
 {
 public Task<IViewComponentResult> InvokeAsync()
 {
 return Task.FromResult<IViewComponentResult>(View("Default"));
 }
 }
}
```

The following code shows the **Default** view component view. It is located in the **Views\Shared\Components\My\Default.cshtml** file.

### A ViewComponent view example

```
some text
```

## Using View Components

To use a view component, you can include the view component in a view. You can include a view component in a view by using the **@Component.InvokeAsync** method. The **@Component.InvokeAsync** method gets the name of the view component class as a parameter. In case the view component class name ends with the suffix **ViewComponent**, then the parameter will be the name of the view component class without the suffix.

The following code shows how to embed the **MyViewComponent** view component in a view named **Index**:

### Using a ViewComponent in a view

```
@await Component.InvokeAsync("My")
```

You can include a view component in a view by:

- using the **@Component.InvokeAsync** method:

```
@await Component.InvokeAsync("My")
```



some text

- using a tag helper:

```
<vc:My></vc:My>
```



some text

The following code shows a simple controller that calls the **Index** view:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
```

```

 return View();
 }
}

```

If a user runs the application and requests the relative URL, /Home/Index, the following text will display in the browser: "some text".

### Invoking a View Component using a Tag Helper

In addition to invoking a view component from a view by using the `@Component.InvokeAsync` method, it is also possible to invoke a view component from a view by using a tag helper. To invoke a view component by using a tag helper, you should use a `vc` element followed by a colon and the name of the view component. To use a view component as a tag helper, you must use the `@addTagHelper` directive, and pass to it the name of the assembly in which the tag helper is declared.

For example, in case the **My** view component was declared in the **ViewComponentExample** assembly, you can invoke it from the **Index** view in the following way:

### Invoking ViewComponent by using a Tag Helper

```

@addTagHelper *, ViewComponentExample
<vc:My></vc:My>

```

Using the `vc:My` element is identical to calling the `@await Component.InvokeAsync("My")` method.

### Invoking a View Component from a Controller

In addition to invoking a view component from a view, you can also invoke a view component from a controller action. To invoke a view component from an action, you can return from the action a **ViewComponentResult** object. To return a **ViewComponentResult** object, you can use the **ViewComponent** method, and pass to it the name of the view component.

The following example shows how to call the **My** view component from an action:

### Invoking ViewComponent from a Controller action

```

public class HomeController : Controller
{
 public IActionResult InvokeVC()
 {
 return ViewComponent("My");
 }
}

```

## Invoking View Components with Parameters

As you saw, the logic of a view component is defined in an **InvokeAsync** method. However, until now you only saw how it is possible to declare the **InvokeAsync** method without passing it parameters. The **InvokeAsync** method can take any number of parameters. Those parameters will be passed when the view component is invoked in a view or in a controller.

The following code shows a view component class named **MyViewComponent**. The **InvokeAsync** method gets a parameter named **param**:

- A view component that gets a parameter:

```
public async Task<IViewComponentResult> InvokeAsync(int param)
{
 int id = await SomeOperationAsync(param);
 return View("Default", id);
}
```

- Pass a parameter to the view component:

```
@await Component.InvokeAsync("My", 5)
```

### Passing a parameter to the **InvokeAsync** method

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ViewComponents
{
 public class MyViewComponent : ViewComponent
 {
 public async Task<IViewComponentResult> InvokeAsync(int param)
 {
 int id = await SomeOperationAsync(param);
 return View("Default", id);
 }

 private async Task<int> SomeOperationAsync(int param)
 {
 await Task.Run(() => /* some operation */);
 return 1;
 }
 }
}
```

The following code shows the **Default** View component view:

### The **Default.cshtml** file

```
@model int
Id: @Model
```

The following code shows how you can pass a parameter from the **Index** view to the view component:

### Passing a parameter from a view to a View component

```
@await Component.InvokeAsync("My", 5)
```

If a user runs the application and requests the relative URL, /Home/Index, the following text will display in the browser: Id: 1.

## Passing Parameters to a View Component using a Tag Helper

It is possible to pass parameters to a view component from a view by using a tag helper. To pass parameters from a view to a view component by using a tag helper, you can add attributes to the **vc** element.

The following code shows how to pass a parameter from the **Index** view to the **My** view component by using a tag helper:

#### **Passing a parameter from a view to a View component by using a Tag Helper**

```
@addTagHelper *, ViewComponentExample
<vc:My param="5"></vc:My>
```

#### **Passing Parameters to a View Component from a Controller**

It is possible to pass parameters to a view component from a controller by using the **ViewComponent** method. The second parameter of the **ViewComponent** method represents an object with properties representing arguments to be passed to the **InvokeAsync** method of the view component.

The following example shows how to pass a parameter from an action to the **My** view component:

#### **Passing a parameter from a Controller to a View component**

```
public class HomeController : Controller
{
 public IActionResult InvokeVC()
 {
 return ViewComponent("My", new { param = 5 });
 }
}
```

### **Demonstration: How to Create and Use View Components**

In this demonstration, you will learn how to create and use view components.

#### **Demonstration Steps**

You will find the steps in the section "Demonstration: How to Create and Use View Components" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_DEMO.md#demonstration-how-to-create-and-use-view-components](https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D_MOD05_DEMO.md#demonstration-how-to-create-and-use-view-components).

# Lab: Developing Views

## Scenario

To construct the user interface of a city's web application, your development team decided to add views. You have been asked to create the views to render a response to a browser.

## Objectives

After completing this lab, you will be able to:

- Add an MVC view to a web application.
- Use Razor to differentiate server-side code from HTML code.
- Write HTML code and tag helpers in a view.
- Add partial views and view components.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD05_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D\\_MOD05\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD05_LAK.md).

## Exercise 1: Adding Views to an MVC Application

### Scenario

To construct the user interface of a web application, views should be added to the web application. In this exercise you will add two views to the web application: **Index** and **Details**. The **Index** view will show a list of cities, and the **Details** view will show the details of a city.

The main tasks for this exercise are as follows:

1. Add a view to show all the cities.
2. Run the application.
3. Add a view to show data for a city.
4. Add a Back link to a view.
5. Add a city name as a link to each city.
6. Run the application.

## Exercise 2: Adding a Partial View

### Scenario

You have been asked to display the population of each city. To do this, you have been asked to add a partial view. In this exercise, you will create a partial view and embed it in the **ShowDataForCity** view.

The main tasks for this exercise are as follows:

1. Add a partial view.
2. Use the partial view in the **ShowDataForCity** view.
3. Run the application.

## Exercise 3: Adding a View Component

### Scenario

Currently, in the **ShowCities** view, for each city, you show a link with the name of the city. You have been asked to show for each city in the **ShowCities** view, the country to which the city belongs and a mini map of the city. To implement this you have been asked to use a view component. In this exercise, you will create a view component and embed it in the **ShowCities** view.

The main tasks for this exercise are as follows:

1. Add a view component class.
2. Add a view component view.
3. Use the view component.
4. Run the application.

**Question:** A member of your team accidentally deleted the **\_ViewImports.cshtml** file. What will be the impact of this deletion?

**Question:** You are asked to replace the tag helper that is used in the **ShowDataForCity.cshtml** file with an HTML helper. The MVC application's behavior should not be changed. What should you do?

# Module Review and Takeaways

In this module, you learned about the pivotal role that views play in the construction of an ASP.NET Core MVC web application. You learned how to build the user interface for your web application by creating views with the Razor view engine. You also learned how partial views and view components can be reused several times in your web application to render similar sections of HTML content on multiple pages. In the next module, you will learn how to create and code models to implement the business logic for your application.

## Review Question

**Question:** You need to have the same block of HTML content on different webpages. Right now, you copy and paste code from one Razor view to others. Do you have a better solution?

## Best Practices

- Use Razor comments, declared with the @\* \*@ delimiters, throughout your Razor views to help explain the view code to other developers in your team.
- Use the @: and <text> tags only when you think that Razor might misinterpret content as code. Razor has sophisticated logic for distinguishing content from code, so this is rarely necessary.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
When a controller tries to access a partial view, an exception is thrown.	Place partial views in the <b>/Views/Shared</b> folder if they need to be used by various controllers.



# Module 6

## Developing Models

### Contents:

Module Overview	06-1
<b>Lesson 1:</b> Creating MVC Models	06-2
<b>Lesson 2:</b> Working with Forms	06-13
<b>Lesson 3:</b> Validating MVC Application	06-24
<b>Lab:</b> Developing Models	06-31
Module Review and Takeaways	06-33

## Module Overview

Most web applications interact with various types of data or objects. An e-commerce application, for example, manages products, shopping carts, customers, and orders. A social networking application might help manage users, status updates, comments, photos, and videos. A blog is used to manage blog entries, comments, categories, and tags. When you write a Model-View-Controller (MVC) web application, you create an MVC model to model the data for your web application. Within this model, you create a model class for each type of object. The model class describes the properties of each type of object and can include business logic that matches business processes. Therefore, the model is a fundamental building-block in an MVC application. In this module, you will learn how to create the code for models.

### Objectives

After completing this module, you will be able to:

- Add a model to an MVC application and write code in it to implement the business logic.
- Use display and edit data annotations.
- Validate user input with data annotations.

# Lesson 1

## Creating MVC Models

An MVC model is a collection of classes. When you create a model class, you define the properties and methods that are required for the kind of object the model class describes. You need to know how to create and describe models, and how to modify the manner in which an MVC creates model class instances when it runs your web application. It is also important to know how controllers pass models to views, and how views can render the data stored in a model to the browser.

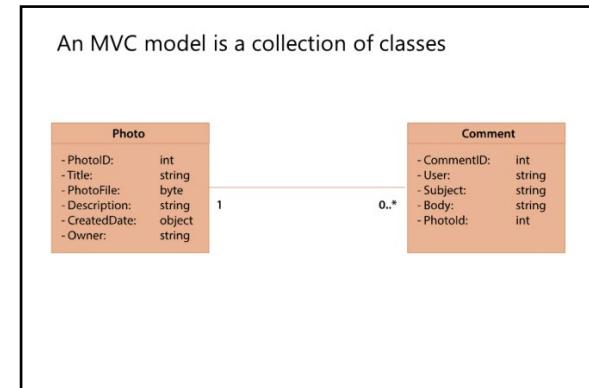
### Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to create MVC models and develop business logic.
- Pass models to views.
- Describe model binders.
- Add create, read, update, and delete (CRUD) operations to controllers.

### Developing Models

Every website presents information about various types of objects. To implement a functional requirement in your web application, you need to define model classes for these objects. If you follow the Agile Development model or Extreme Programming, you begin with a simple definition of the class—perhaps its name and a few properties. Then, you discuss with users and add details to the planned model class including the complete set of properties and its relationships to other classes. When you start developing the model, you can refer to use cases or user stories to ensure that these model details are correct.



**Note:** Agile Development model and Extreme Programming are covered in Module 2, "Designing ASP.NET Core MVC Web Applications".

After you have fully understood the requirements for a model, you can write model classes to implement these requirements. Programmers should create their models in a folder named **Models**. Because MVC relies on convention more than configuration, we recommend adhering to the conventions.

The following lines of code illustrate how to create a model class named **Photo**:

#### The Photo model class

```

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

```

```

 public string Owner { get; set; }

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

```

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

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

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

#### The Comment model class

```

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

```

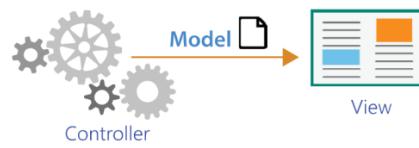
Notice that the **Comment** class includes a **PhotoID** property. This property stores the ID of the photo that the user commented on, and it ties the comment to a single photo. Also, notice that the **Comment** class includes a **Photo** property, which returns the **Photo** object that the comment relates to. These properties implement the other side of the relationship between photos and comments.

## Passing Models to Views

When an ASP.NET Core MVC web application receives a request from a web browser, it instantiates a controller object to respond to the request. Next, the ASP.NET Core MVC web application determines the action method that should be called on the controller object. Often, the action creates a new instance of a model class, which can be passed to a view to display results to the user.

 **Best Practice:** Typically, most simple examples pass the model objects to the view. However, for large applications, this is generally not a best practice. For large applications, we recommend that you use **ViewModels** to separate the presentation from the domain.

The instances of model classes are usually created in a controller and passed to a view



The following code example shows a simple model class:

### A simple model class

```
namespace ModelNamespace
{
 public class SomeModel
 {
 public string Text { get; set; }
 }
}
```

The following code shows a controller that creates a model and passes it to a view:

### A controller that passes a model to a view

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 SomeModel model = new SomeModel() { Text = "some text" };
 return View(model);
 }
}
```



**Note:** Controllers are covered in Module 4, "Developing Controllers".

You can use the Razor **@model** operator to specify the strongly typed domain model type of the view. To access the value of a property in the domain object, use **@Model.PropertyName**.

The following code shows a view that receives a model of the **SomeModel** type and uses it to generate a response:

### A view that uses a model

```
@model ModelNamespace.SomeModel
@Model.Text
```

If a user requests the relative URL, /Home/Index, the following string displays on the screen: some text.



**Note:** Views are covered in Module 5, "Developing Views".

## Passing a Collection of Items

Some views display several instances of a model class. For example, a product catalog page displays several instances of the **Product** model class. In such cases, the controller passes a list of model objects to the view, instead of a single model object. You usually loop through the items in the list by using a Razor **foreach** loop.

The following code shows a controller that creates a list of items and passes them to a view:

### A controller that passes a list of items to a view

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
```

```

 SomeModel item1 = new SomeModel() { Text = "first item" };
 SomeModel item2 = new SomeModel() { Text = "second item" };
 List<SomeModel> items = new List<SomeModel>() { item1, item2 };
 return View(items);
 }
}

```

The following code shows a view that receives a collection of items and uses it to generate a response:

### A view that receives a collection of items

```

@model IEnumerable<ModelNamespace.SomeModel>

@foreach (var item in @Model)
{
 <div>@item.Text</div>
}

```

If a user requests the relative URL, /Home/Index, the following text will be displayed in the browser:

### Text displayed in browser

```

first item
second item

```

## Passing a Model to a Different View

The **Controller.View** method is overloaded. The preceding examples used a version of this method that got only an instance of a model as a parameter. When you use this version of the **Controller.View** method, you don't have to specify the name of the view because it is implicitly identical to the name of the action. However, you can use another version of the **Controller.View** method to call a view whose name is different from the name of the action. When you use this version, you need to explicitly specify the name of the view.

The following code shows a controller that creates a model and passes it to a view. The name of the action is **Index**, while the name of the view is **Display**:

### Passing a model to the Display view

```

public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 SomeModel model = new SomeModel() { Text = "some text" };
 return View("Display", model);
 }
}

```

The following code shows a view that receives a model of the **SomeModel** type and uses it to generate the response:

### The Display.cshtml file

```

@model ModelNamespace.SomeModel

@Model.Text

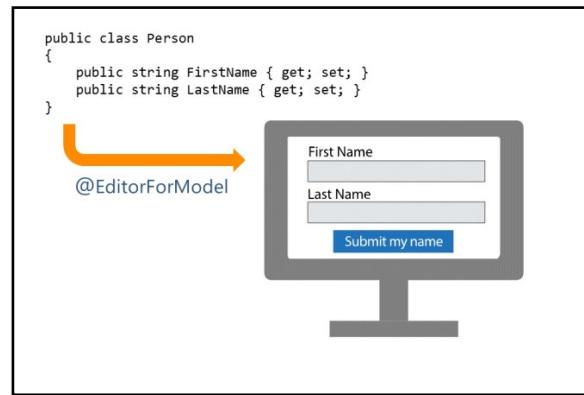
```

If a user requests the relative URL, /Home/Index, the following string displays on the screen: some text.

## Binding Views to Model Classes and Displaying Data

The user interface is a vital component of any system because it is the part that the users, budget holders, and other stakeholders see and interact with most. Users expect a user interface that is well designed. As a developer, you have an opportunity to impress your users by designing and implementing a sophisticated user interface. In an ASP.NET Core MVC application, you use views to define the user interface.

The views that are designed to display properties from a specific model class are called *strongly typed views*. You can bind such views to the model class whose properties they are displaying, which enables you to get IntelliSense feedback as you write the Razor code. Other views might display properties from different model classes in different cases, or may not use a model class at all. These types of views are called *dynamic views* and you cannot bind them to a specific model class. It is important to understand how to write Razor code for both these views.



### Strongly Typed Views

Often, when you create a view, the controller action will pass an object of a specific model class. For example, if you are writing the **Display** view for the **Display** action in the **Product** controller, you know from the action code that the action will always pass a **Product** object to the view.

In such cases, you can create a strongly typed view. A strongly typed view includes a declaration of the model class. When you declare the model class in the view, Microsoft Visual Studio provides additional IntelliSense feedback and error-checking as you write the code because it can check the properties of the model class. This also simplifies troubleshooting run-time errors. Therefore, Whenever you can, create strongly typed views because this extra IntelliSense and error-checking features can help you reduce coding errors. A strongly typed view only works with an object of the model class in the declaration.

In the view files, you can access properties of the model object by using the **Model** keyword. To access a property in a model, use: **@Model.PropertyName**.

### Using Dynamic Views

Sometimes, you might want to create a view that can display more than one model class. For example, you might have a model class named **Product** for your own products and another model class named **ThirdPartyProduct** for products from other suppliers. You want to create a view that can display your own products and the third-party products together. These model classes might have some similar properties while others might differ. For example, both **Product** and **ThirdPartyProduct** might include the **Price** property, while only the **ThirdPartyProduct** model class includes the **Supplier** property.

Furthermore, sometimes, the controller action does not pass any model class to the view. The site home page is the most common example of such a view.

In such cases, you can create a dynamic view. A dynamic view does not include the **@model** declaration at the top of the page. You can later choose to add the **@model** declaration to change the view into a strongly typed view.

When you create dynamic views, Visual Studio does not provide much IntelliSense feedback and error checking because it cannot check the model class properties to verify the code. In such scenarios, it is your responsibility to ensure that you access only the properties that exist. To access a property that may or may not exist, such as the **ThirdPartyProduct.Supplier** property in the preceding example, check the property for **null** before you use it.

## Using the `@Html.EditorForModel` HTML Helper

In many cases, the model is the basis for building the user interface. A view can render the value of model's properties by using the `@Model` directive. However, there are several HTML helpers and tag helpers that you can use to simplify working with models in views. For example, the `@Html.EditorForModel` HTML helper returns an HTML input element for each property in the model.

 **Note:** Other HTML helpers and tag helpers will be covered in Lesson 2, "Working with Forms".

The following code shows a simple model class:

### A Model class

```
namespace ModelNamespace
{
 public class Person
 {
 public string FirstName { get; set; }
 public string LastName { get; set; }
 }
}
```

The following code shows a simple controller class:

### A Controller class

```
public class PersonController : Controller
{
 [Route("Person/GetName")]
 public IActionResult GetName()
 {
 return View();
 }
}
```

The following code shows how to use the `@html.EditorForModel` HTML helper in a view to display all properties of a model:

## Using the `@html.EditorForModel` HTML Helper

```
@model ModelNamespace.Person

<form action="/Person/GetName" method="post">
 @Html.EditorForModel()
 <input type="submit" value="Submit my name" />
</form>
```

If a user requests the relative URL, /Person/GetName, a form displays in the browser. The form contains two text boxes, one of them represents the **FirstName** property of the model and the other one represents the **LastName** property of the model, and a **submit** button. If a user enters values in the text boxes and clicks the **submit** button, a postback occurs to the **GetName** action in the **PersonController** controller.

## Using the `@Html.BeginForm` HTML Helper

In the previous example, a form was written explicitly in the view. When working with ASP.NET Core MVC, Using the `@Html.BeginForm` HTML helper is considered as a better approach.

The following code shows how to use the **@Html.BeginForm** HTML helper in a view to render a form implicitly:

### Using the **@html.BeginForm** HTML Helper

```
@model ModelNamespace.Person

@using (Html.BeginForm())
{
 @Html.EditorForModel()
 <input type="submit" value="Submit my name" />
}
```



**Note:** The **@html.BeginForm** HTML helper will be covered in Lesson 2, "Working with Forms".

## Demonstration: How to Bind Views to Model Classes

In this demonstration, you will learn how to add a model to an MVC application and how to pass the model to views. You will explore different ways to render the model properties from the views to the browser.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Bind Views to Model Classes" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_DEMO.md#demonstration-how-to-bind-views-and-model-classes](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD06_DEMO.md#demonstration-how-to-bind-views-and-model-classes)

## What Are Model Binders?

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

- The ASP.NET Core MVC runtime uses model binders to determine how parameters are passed to actions
- The default model binder passes parameters by using the following logic:
  - The binder examines the definition of the action that it must pass parameters to
  - The binder searches for values in the request that can be passed as parameters

### What Does a Model Binder Do?

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

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

To understand the default model binding process, consider the following sequence of steps:

1. A web browser receives the following request: <http://www.contoso.com/product/display/45>.
2. This request identifies three aspects:
  - o The model class that interests the user. The user has requested a product.
  - o The operation to perform on the model class. The user has requested that the product be displayed.
  - o The specific instance of the model class. The user has requested that the product with ID 45 be displayed.
3. The request is received by the ASP.NET Core MVC runtime.
4. The ASP.NET Core MVC runtime calls an action and passes the correct parameters to it. In the example, the ASP.NET Core MVC runtime calls the **Display** action in the **Product** controller and passes the ID **45** as a parameter to the action, so that the right product can be displayed. To do this, the ASP.NET Core MVC runtime uses model binders to determine how parameters are passed to actions.

## How the Default Model Binder Passes Parameters

In a default ASP.NET Core MVC application, there is only one model binder to use – the default model binder. The default model binder passes parameters by using the following logic:

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

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

## Handling Form Values

You can overload an action method. You can create two action methods with the same name that differ by the parameters they get. A common scenario is an action that loads a form and an action that saves a form, both of them have the same name. The ASP.NET Core MVC runtime knows which method should be invoked by using the **HttpPostAttribute** attribute and **HttpGetAttribute** attribute. You can annotate an action with the **HttpGetAttribute** attribute, which indicates that it is a GET operation that is usually used to load a form. You can also annotate an action with the **HttpPostAttribute** attribute, which indicates that it is a POST operation that is usually used to save a form.

The following code shows a simple model class:

### A Model class

```
namespace ModelNamespace
{
 public class Person
 {
 public string FirstName { get; set; }
 public string LastName { get; set; }
 }
}
```

The following code shows a controller class that contains two action methods, both of them with the name **GetName**. One of them is annotated with the **HttpGetAttribute** attribute, and the other one is annotated with the **HttpPostAttribute** attribute:

### Using **HttpGetAttribute** and **HttpPostAttribute**

```
public class PersonController : Controller
{
 [Route("Person/GetName")]
 [HttpGet]
 public IActionResult GetName()
 {
 return View();
 }

 [Route("Person/GetName")]
 [HttpPost]
 public IActionResult GetName(Person person)
 {
 return View("ShowName", person);
 }
}
```

The following code shows the view that is called by the **GetName()** action:

### The **GetName.cshtml** file

```
@model ModelNamespace.Person

@using (Html.BeginForm())
{
 @Html.EditorForModel()
 <input type="submit" value="Submit my name" />
}
```

The following code shows the view that is called by the **GetName(Person person)** action:

### The **ShowName.cshtml** file

```
@model ModelNamespace.Person

Hello @Model.FirstName @Model.LastName
```

If a user requests the relative URL, /Person/GetName, the **GetName()** action is called. The **GetName()** action calls the **GetName** view. The **GetName** view sends to the browser a form that contains two text boxes and a submit button. When a user fills the text boxes and clicks the submit button, the ASP.NET Core MVC runtime uses the model binder to create an instance of type **Person** and fills its properties based on what the user entered in the form. Then, the ASP.NET Core MVC runtime calls the **GetName(Person person)** action, passing it the instance of type **Person**. The **GetName(Person person)**

action calls the **ShowName** view. The **ShowName** view sends the values of the properties of the **Person** model to the browser.

## Adding CRUD Operations to Controllers

Often, a controller has CRUD operations. The create operations are used to create or add new items. The read operations are used to read, retrieve, search, or view existing entries. The update operations are used to update or edit existing entries. The delete operations are used to delete existing entries.

The following example shows a controller with CRUD operations:

A controller might be associated with CRUD operations:

- Create operations: Used to create or add new items
- Read operations: Used to read, retrieve, search or view existing entries
- Update operations: Used to update or edit existing entries
- Delete operations: Used to delete existing entries

### A controller with CRUD operations

```
public class PersonController : Controller
{
 [HttpGet]
 public IActionResult Index()
 {
 // TODO: Add logic here
 return View(people);
 }

 [HttpGet]
 public IActionResult Details(int id)
 {
 // TODO: Add logic here
 return View(person);
 }

 [HttpGet]
 public IActionResult Create()
 {
 // TODO: Add logic here
 return View();
 }

 [HttpPost]
 public IActionResult Create(Person person)
 {
 // TODO: Add logic here
 return RedirectToAction("Index");
 }

 [HttpGet]
 public IActionResult Edit(int id)
 {
 // TODO: Add logic here
 return View(person);
 }

 [HttpPost]
 public IActionResult Edit(int id, Person person)
 {
 // TODO: Add logic here
 return RedirectToAction("Index");
 }
}
```

```
[HttpGet]
public IActionResult Delete(int id)
{
 // TODO: Add logic here
 return View(person);
}

[HttpPost, ActionName("Delete")]
public IActionResult DeleteConfirmed(int id)
{
 // TODO: Add logic here
 return RedirectToAction("Index");
}
```

In the preceding example, there are two read operations: the **Index** method returns a collection of people, while the **Details** method returns a specific person. There are two create operations: the **Create()** method allows the user to enter the properties of the person to create and the **Create(Person person)** method creates a person based on the properties entered by the user. There are two update operations: the **Edit(int id)** method allows the user to change the properties of an existing person, while the **Edit(int id, Person person)** method saves the changes made by the user to the person. There are two delete operations accepting the same parameter *id*. Since the two delete operations have the same name and parameter, and it is illegal to have two methods in c# with the same signature, the name of the second action is **DeleteConfirmed**.

## Lesson 2

# Working with Forms

When you create a model class, you define the properties and methods that are appropriate for the kind of object the model class describes. Based on the class properties and methods, MVC can determine how to render the model on a webpage.

The ASP.NET Core MVC Framework includes many HTML helper functions and tag helpers that you can use in views. You can use helpers to render values, labels, and input controls such as text boxes. Helpers include common logic that makes it easier for you to render HTML for MVC model class properties and other tasks. You can also create custom helpers. You need to know the appropriate helper to use in any scenario.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use the display and edit data annotations.
- Use HTML helpers and tag helpers to display a value of a model class property.
- Use HTML helpers and tag helpers to render an editor control for a model class property.
- Use various HTML helpers and tag helpers to build a user interface for a web application.

### Using Display and Edit Data Annotations

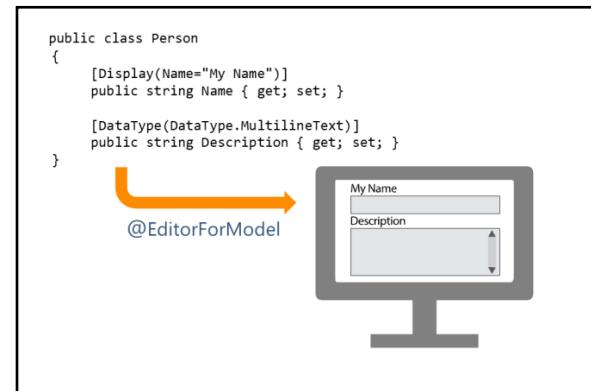
A model class usually contains several properties each of which usually includes:

- The name of the property such as **EmailAddress**.
- The data type of the property such as **string**.
- The access levels of the property such as the **get** and **set** keywords to indicate read and write access.

In addition, you can supply more metadata to describe the properties of models in ASP.NET Core MVC. The ASP.NET Core MVC runtime uses this metadata to determine how to render each property in views for displaying and editing. These attributes are called *display and edit annotations*.

For example, property names in C# cannot contain spaces. On a rendered webpage, you may often want to include spaces in a property label. For example, you may want to render a property called **EmailAddress** with the label **Email Address**. To provide MVC with this information, you can use the **DisplayAttribute** attribute.

Sometimes, you might want to provide additional type information for a property. For example, a property with a name **Password** has a data type **string**. However, it is probably important to mark this property as a password to inform the browser that it needs to show an asterisk or circle when a user types a character. To provide MVC with this information, you can use the **DataTypeAttribute** attribute.



The following code shows how you can annotate the properties of a model with the **DisplayAttribute** and the **DataTypeAttribute** attributes:

### Using data annotation

```
public class Person
{
 [Display(Name="My Name")]
 public string Name { get; set; }

 [DataType(DataType.Password)]
 public string Password { get; set; }

 [DataType(DataType.Date)]
 public DateTime Birthdate { get; set; }

 [Display(Name="Email Address")]
 public string EmailAddress { get; set; }

 [DataType(DataType.MultilineText)]
 public string Description { get; set; }
}
```

The following code shows a simple controller class named **PersonController**:

### A Controller example

```
public class PersonController : Controller
{
 [Route("Person/GetDetails")]
 public IActionResult GetDetails()
 {
 return View();
 }
}
```

The following code illustrates how to display the properties of the model by using the **Html.EditorForModel** HTML helper:

### The GetDetails view

```
@model ModelNamespace.Person

@Html.EditorForModel()
```

If a user requests the relative URL, /Person/GetDetails, a form displays in the browser. The form contains the following elements:

- A label with the text “My Name” and a text box below it. Notice that although the name of the property is **Name**, in the browser the text “My Name” displays because the **Name** property has a **DisplayAttribute** attribute.
- A label with the text “Password” and a text box below it. If a user enters characters in the text box, they appear as asterisks or circles in the browser because the **Password** property has a **DataTypeAttribute** attribute with a **Password** data type.
- A label with the text “Birthdate” and a text box below it. The text box is an **input** HTML element with a **type=“date”** attribute because the **Birthdate** property has a **DataTypeAttribute** attribute with a **Date** data type.
- A label with the text “Email Address” and a text box below it. Notice that although the name of the property is **EmailAddress**, in the browser the text “Email Address” displays because the **EmailAddress** property has a **DisplayAttribute** attribute.

- A label with the text “Description” and a text area below it because the **Description** property has a **DataTypeAttribute** attribute with a **MultilineText** data type.

 **Note:** Notice that the data annotations that are mentioned above are included in the **System.ComponentModel.DataAnnotations** namespace.

## Using Display Helpers

HTML helpers are simple methods that typically return a string. This string is a small section of HTML that the view engine can insert into the view file to generate the completed webpage. You can write views that render any type of HTML content without using a single helper if you prefer not to use HTML helpers. However, HTML helpers simplify the task of managing HTML content by rendering common HTML strings for frequently used scenarios.

• **Html.DisplayNameFor()**

```
@Html.DisplayNameFor(model => model.FirstName)
```



First Name:

• **Html.DisplayFor()**

```
@Html.DisplayFor(model => model.FirstName)
```



James

 **Note:** The **Html.ActionLink** and **Url.Action** HTML helpers were introduced in Module 5, “Developing Views”. In this module, other HTML helpers are covered.

MVC includes several helpers that display properties from model classes. You can use these helpers to build views that display product details, comment details, user details, and so on. **Html.DisplayNameFor** renders the name of a model class property. **Html.DisplayFor** renders the value of a model class property. Both these helpers examine the definition of the property in the model class, including the data display annotations, to ensure that they render the most appropriate HTML.

### The **Html.DisplayNameFor** Helper

You can use the **Html.DisplayNameFor** helper to render the display name for a property from the model class. If your view is strongly typed, Visual Studio checks whether the model class contains a property with the correct name as you type the code. Otherwise, you must ensure that you use a property that exists or verify that it is not **null** before you use it. You specify the property of the model class to the **Html.DisplayNameFor** HTML helper by using a lambda expression.

The following code shows a model class named **Person**:

#### A Model class

```
public class Person
{
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public bool ContactMe { get; set; }
}
```

The following code shows a controller named **PersonController** that creates an instance of the **Person** model and passes it to a view named **ShowDetails**:

### A Controller class

```
public class PersonController : Controller
{
 [Route("Person/ShowDetails")]
 public IActionResult ShowDetails()
 {
 Person model = new Person() { FirstName = "James", LastName = "Smith", ContactMe
= true };
 return View(model);
 }
}
```

The following code example shows a view named **ShowDetails** that renders the display name of the properties of the **Person** model by using the **Html.DisplayNameFor** HTML helper:

### Using the **Html.DisplayNameFor** helper

```
@model ModelNamespace.Person

<h1>Person properties</h1>
@Html.DisplayNameFor(model => model.FirstName)

@Html.DisplayNameFor(model => model.LastName)

@Html.DisplayNameFor(model => model.ContactMe)
```

If a user requests the relative URL "/Person/ShowDetails" the following will be displayed in the browser:

### Browser display

```
Person properties
FirstName
LastName
ContactMe
```

The text rendered by the **Html.DisplayNameFor** helper depends on the model class. If you used a **DisplayAttribute** attribute to give a more descriptive name to a property, **Html.DisplayNameFor** will use the value of the *Name* parameter. Otherwise, it will render the name of the property.

The following code shows a model class named **Person** that has data annotations:

### A Model class with data annotations

```
public class Person
{
 [Display(Name="First Name: ")]
 public string FirstName { get; set; }

 [Display(Name="Last Name: ")]
 public string LastName { get; set; }

 [Display(Name="Contact me? ")]
 public bool ContactMe { get; set; }
}
```

If a user requests the relative URL "/Person>ShowDetails", the following will be displayed in the browser:

### Browser display

#### Person properties

First Name:  
Last Name:  
Contact me?

### The `Html.DisplayFor` Helper

The `Html.DisplayFor` helper considers any display annotations that you specify in the model class, and then renders the value of the property. It generates different HTML markups depending on the data type of the property that is being rendered. For example, if the property is of type `bool`, it renders an HTML input element for a check box.

The following code example shows a view named `ShowDetails` that uses the `Html.DisplayFor` HTML helper:

### Using the `Html.DisplayFor` helper

```
@model ModelNamespace.Person

<h1>Person details</h1>
@Html.DisplayNameFor(model => model.FirstName)
@Html.DisplayFor(model => model.FirstName)

@Html.DisplayNameFor(model => model.LastName)
@Html.DisplayFor(model => model.LastName)

@Html.DisplayNameFor(model => model.ContactMe)
@Html.DisplayFor(model => model.ContactMe)
```

If a user requests the relative URL "/Person>ShowDetails", the following will be displayed in the browser:

### Browser display

#### Person details

First Name: James  
Last Name: Smith  
Contact me?

## Using Editor Helpers

Within HTML forms, there are many HTML input controls that you can use to gather data from users. In Razor views, the `Html.LabelFor` and `Html.EditorFor` HTML helpers make it easy to render the most appropriate input controls for the properties of a model class. To understand how these helpers render HTML, you must first understand the HTML input controls. The following table describes some common HTML controls.

#### • `Html.LabelFor()`

```
@Html.LabelFor(model => model.ContactMe)

L <label for="ContactMe">
 Contact Me
</label>
```

#### • `Html.EditorFor()`

```
@Html.EditorFor(model => model.ContactMe)

L <input type="checkbox"
 name="ContactMe" />
```

Control	Example	Description
Text box	<pre>&lt;input type="text" name="Title" /&gt;</pre>	Renders a single-line text box in which the user can enter a string. The <b>name</b> attribute is used to identify the entered value in the query string or to send form data by using the POST method.
Multi-line text box	<pre>&lt;textarea name="Description" rows="20" cols="80" /&gt;</pre>	Renders a multi-line text box in which the user can enter longer strings.
Check box	<pre>&lt;input type="checkbox" name="ContactMe" /&gt;</pre>	Renders a check box to submit a boolean value.

## The **Html.LabelFor** HTML Helper

The **Html.LabelFor** helper is similar to the **Html.DisplayNameFor** helper because it renders the name of the property that you specify, considering the **DisplayAttribute** attribute if it is specified in the model class. However, unlike the **Html.DisplayNameFor** helper, the **Html.LabelFor** helper renders a **<label>** element.

## The **Html.EditorFor** HTML Helper

The **Html.EditorFor** helper renders the most appropriate HTML input elements and other form controls for each property data type in a model class. For example, the **Html.EditorFor** helper renders **<input type="text">** for a string property. If the string property is annotated with **[DataType(DataType.MultilineText)]**, the **Html.EditorFor** helper renders a **<textarea>** element instead. The following table describes the HTML that **Html.EditorFor** renders for different model class properties.

Control	Model Class Property	HTML Rendered by EditorFor()
Text box	<pre>public string Title { get; set; }</pre>	<pre>&lt;input type="text" name="Title" /&gt;</pre>
Multi-line text box	<pre>[DataType(DataType.MultilineText)] public string Description { get; set; }</pre>	<pre>&lt;textarea name="Description" rows="20" cols="80" /&gt;</pre>
Check box	<pre>public bool ContactMe { get; set; }</pre>	<pre>&lt;input type="checkbox" name="ContactMe" /&gt;</pre>

If the action passes an existing model object to the view, the **Html.EditorFor** helper also populates each control with the current values of each property.

The following code shows a model class named **Person** that has data annotations:

#### A Model class with data annotations

```
public class Person
{
 [Display(Name="First Name")]
 public string FirstName { get; set; }

 [Display(Name="Last Name")]
 public string LastName { get; set; }

 [Display(Name="Contact me?")]
 public bool ContactMe { get; set; }
}
```

The following code shows a controller named **PersonController**:

#### A Controller class

```
public class PersonController : Controller
{
 [Route("Person/GetDetails")]
 public IActionResult GetDetails()
 {
 return View();
 }
}
```

The following code shows a view named **GetDetails** that uses the **Html.LabelFor** HTML helper and the **HTML.EditorFor** HTML helper:

#### Using the **Html.LabelFor** and **HTML.EditorFor** HTML helpers

```
@model ModelNamespace.Person

<h1>Get person details</h1>
@Html.LabelFor(p => p.FirstName)
@Html.EditorFor(p => p.FirstName)

@Html.LabelFor(p => p.LastName)
@Html.EditorFor(p => p.LastName)

@Html.LabelFor(p => p.ContactMe)
@Html.EditorFor(p => p.ContactMe)
```

If a user requests the relative URL "/Person/GetDetails", the following will be displayed in the browser:

#### Browser display

**Get person details**

First Name:

Last Name:

Contact me?

#### The **LabelTagHelper** and **InputTagHelper** Tag Helpers

Tag helpers are an alternative to HTML helpers. Similar to HTML helpers, the role of tag helpers is to embed server-side code inside a view. However, it is much easier for non-programmers, such as designers, to understand them because tag helpers look like regular HTML elements. Tag helpers are implemented by using programming languages such as C#.

 **Note:** In Module 5, "Developing Views", the **AnchorTagHelper** tag helper was introduced.

The **LabelTagHelper** tag helper is an alternative to the **Html.LabelFor** HTML helper. It generates a **<label>** element for a property in a model and you can use it by adding an **asp-for** attribute to a **<label>** element. The value of the generated **for** HTML attribute matches the name of the property in the model class. The content of the **<label>** element matches the **Name** property of the **DisplayAttribute** attribute that is specified in the model class.

The **InputTagHelper** tag helper is an alternative to the **Html.EditorFor** HTML helper. It generates an **<input>** element for a property of a model and you can use it by adding an **asp-for** attribute to a **<input>** element. The **InputTagHelper** tag helper adds an **id** and a **name** based on the property name specified in the **asp-for** attribute. Similar to the **Html.Editor** HTML helper, the type of the **input** element is determined based on the .NET type of the property of the model class. For example, if the .NET type of the property is **string**, the type of the input element is **text**, while if the .NET type of the property is **bool**, the type of the input element is **checkbox**.

The following code shows how you can rewrite the GetDetails view to use tag helpers instead of using HTML helpers:

### Using Tag Helpers

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@model ModelNamespace.Person

<h1>Get person details</h1>
<label asp-for="FirstName"></label>
<input asp-for="FirstName" />

<label asp-for="LastName"></label>
<input asp-for="LastName" />

<label asp-for="ContactMe"></label>
<input asp-for="ContactMe" />
```

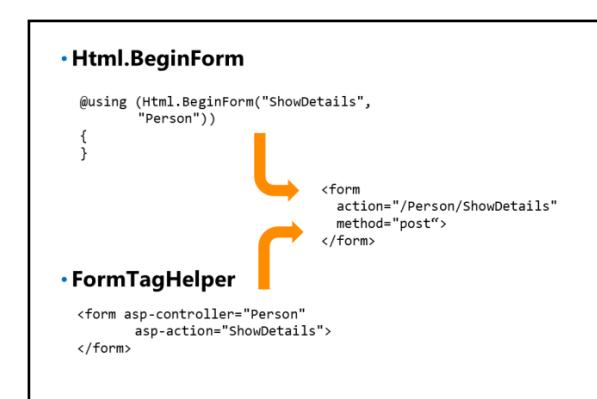
If a user requests the relative URL "/Person/GetDetails" the following will be displayed in the browser:

#### Browser display

**Get person details**  
 First Name:   
 Last Name:   
 Contact me?

## Using Form Helpers

To accept user input, you can provide a form on your webpage. A typical form consists of a set of labels and input controls. The labels indicate to the user the property for which they should provide a value. The input controls enable the user to enter a value. Input controls can be text boxes, check boxes, radio buttons, file selectors, drop-down lists, or other types of control. There is usually a submit button and cancel button on forms.



## The `Html.BeginForm` HTML Helper

To build a form in HTML, you must start with a `<form>` element on the HTML webpage and all labels and input controls must be within the `<form>` element. In an MVC view, you can use the **`Html.BeginForm`** HTML helper to render this element and set the controller action to which the form sends data.

You can also specify the HTTP method that the form uses to submit data. If the form uses the POST method, which is the default, the browser sends form values to the web server in the body of the form. If the form uses the GET method, the browser sends form values to the web server in the query string in the URL.

In the rendered HTML, the `<form>` element must be closed with a `</form>` tag. In Razor views, you can ensure that the form element is closed with a `@using` code block. Razor renders the `</form>` tag at the closing bracket of the code block.

The following code shows a model class named **Person** that has data annotations:

### A Model class with data annotations

```
public class Person
{
 [Display(Name="First Name: ")]
 public string FirstName { get; set; }

 [Display(Name="Last Name: ")]
 public string LastName { get; set; }

 [Display(Name="Contact me?")]
 public bool ContactMe { get; set; }
}
```

The following code shows a controller named **PersonController**:

### A Controller class

```
public class PersonController : Controller
{
 [Route("Person/GetDetails")]
 public IActionResult GetDetails()
 {
 return View();
 }

 [Route("Person>ShowDetails")]
 public IActionResult ShowDetails(Person person)
 {
 return View(person);
 }
}
```

The following code example shows how to render a form that sends data to the **ShowDetails** action in the **PersonController** controller. The form uses the POST HTTP verb:

### Using the `Html.BeginForm` helper

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@model ModelNamespace.Person

<h1>Get person details</h1>
@using (Html.BeginForm("ShowDetails", "Person"))
{
 <label asp-for="FirstName"></label>
 <input asp-for="FirstName" />


```

```

<label asp-for="LastName"></label>
<input asp-for="LastName" />

<label asp-for="ContactMe"></label>
<input asp-for="ContactMe" />

<input type="submit" value="Submit my details" />
}

```

The following code example shows a view named **ShowDetails**:

### The ShowDetails view

```

@model ModelNamespace.Person

<h1>Person details</h1>
@Html.DisplayNameFor(model => model.FirstName)
@Html.DisplayFor(model => model.FirstName)

@Html.DisplayNameFor(model => model.LastName)
@Html.DisplayFor(model => model.LastName)

@Html.DisplayNameFor(model => model.ContactMe)
@Html.DisplayFor(model => model.ContactMe)

```

If a user requests the relative URL “/Person/GetDetails” a form displays in the browser. If a user enters his first name and last name and clicks the submit button, the **ShowDetails** action of the **PersonController** controller is called.

 **Note:** The **Html.BeginForm** HTML helper is overloaded. For example, if you would like to specify the HTTP method that the form uses to submit data, you can enter **Html.BeginForm("ShowDetails", "Person", FormMethod.Post)**.

### The FormTagHelper Tag Helper

The **FormTagHelper** tag helper is an alternative to the **Html.BeginForm** HTML helper. It generates a **<form>** HTML element. You can add to the **FormTagHelper** helper an **asp-controller** attribute to bind it to a specific controller. You also can add to the **FormTagHelper** helper an **asp-action** attribute to bind it to a specific action. Although the default form method value is post, you can specify another form method.

The following code shows how the **GetDetails** view can be rewritten to use the **FormTagHelper** tag helper instead of using the **Html.BeginForm** HTML helper.

### Using FormTagHelper

```

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@model ModelNamespace.Person

<h1>Get person details</h1>
<form asp-controller="Person" asp-action="ShowDetails">
 <label asp-for="FirstName"></label>
 <input asp-for="FirstName" />

 <label asp-for="LastName"></label>
 <input asp-for="LastName" />

 <label asp-for="ContactMe"></label>
 <input asp-for="ContactMe" />

 <input type="submit" value="Submit my details" />

```

```
</form>
```

## Demonstration: How to Use Display and Edit Data Annotations

In this demonstration, you will learn how to use display and edit data annotations. Then, you will learn how to build a form in HTML by using form helpers and editor helpers. Finally, you will learn how to display in the browser the model properties by using display helpers.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Use Display and Edit Data Annotations” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_DEMO.md#demonstration-how-to-use-display-and-edit-data-annotations](https://github.com/MicrosoftLearning/20486D-DevelopingASPMVCWebApplications/blob/master/Instructions/20486D_MOD06_DEMO.md#demonstration-how-to-use-display-and-edit-data-annotations)

## Lesson 3

# Validating MVC Application

It is very important to validate user input in web applications. The validation of user input might be a challenging task. The developer needs to write code that checks for errors made by a user when he fills a form. ASP.NET Core MVC enables the developer to ease the process of input validation. When you create a model class in ASP.NET Core MVC, you can annotate the class properties with attributes that determine how to validate user input. There are several HTML helpers and tag helpers that you can use in views to validate user input data.

### Lesson Objectives

After completing this lesson, you will be able to:

- Validate user input with data annotations.
- Use HTML helpers and tag helpers to validate data entered by users.
- Use custom validation in MVC applications.

### Validating User Input with Data Annotations

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

- *Name*. This is required input. The user must enter some data in this field.
- *Age*. This must be an integer between 0 and 150.
- *Email Address*. This is required input. The value entered must be a valid email address.
- *Description*. This must be a string no longer than 20 characters.

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

 [Range(0, 150)]
 public int Age { get; set; }

 [Required]
 [RegularExpression(".+\\@.+\\..+")]
 public string EmailAddress { get; set; }

 [DataType(DataType.MultilineText)]
 [StringLength(20)]
 public string Description { get; set; }
}
```

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

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

#### Using validation data annotations

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

 [Range(0, 150)]
 public int Age { get; set; }

 [Required]
 [RegularExpression(".+\\@.+\\..+")]
}
```

```

 public string EmailAddress { get; set; }

 [DataType(DataType.MultilineText)]
 [StringLength(20)]
 public string Description { get; set; }
}

```

The **Required**, **Range**, **StringLength**, and **RegularExpression** annotations implement input validation in MVC. If users do not enter data that satisfies the criteria specified for each property, the view displays a standard error message that prompts the user to enter the correct data.

In the preceding example, you can see that the user must enter a name. To specify the error message that the user sees when data is not valid, use the **ErrorMessage** property on the validation data annotations. The user must also insert an age that will be an integer number between 0 and 150. For the **EmailAddress** property, the user must enter a value that matches the regular expression. The regular expression in the example is a simple expression that requires an @ symbol and a dot. For the **Description** property, the user is not allowed to insert a string that contains more than 20 characters.

### Using ModelState.IsValid in the Controller

A controller can use the **ModelState.IsValid** property to check whether the user has submitted valid data. If the data is valid, value of **ModelState.IsValid** will be **true**, otherwise it will be **false**.

The following code shows a controller that uses the **ModelState.IsValid** property to check if a user filled all data of a person correctly:

### Using the ModelState.IsValid property

```

public class PersonController : Controller
{
 [Route("Person/GetDetails")]
 [HttpGet]
 public IActionResult GetDetails()
 {
 return View();
 }

 [Route("Person/GetDetails")]
 [HttpPost]
 public IActionResult GetDetails(Person person)
 {
 if (ModelState.IsValid)
 {
 return View("ShowDetails", person);
 }

 else
 {
 return View();
 }
 }
}

```

If the data is valid, a view named **ShowDetails** is called. Otherwise, the **GetDetails** view is called again, so that the user can correct the invalid data.

The following code shows the code of the **GetDetails** view:

### The GetDetails view

```
@model ModelNamespace.Person

<h1>Get person details</h1>
@using (Html.BeginForm())
{
 @Html.EditorForModel()
 <input type="submit" value="Submit my details" />
}
```

When a user requests the relative URL "/Person/GetDetails" a form appears in the browser without any error messages. However, if a user enters data that is not valid and clicks the button, the form that is rendered from the **GetDetails** view will contain error messages next to the fields that were not filled correctly by the user. For example, if a user did not specify his name, the message "Please enter a name." appears next to the **Name** text box. The reason is that the **@Html.EditorForModel** HTML helper renders the error messages to the browser.

 **Note:** The example above demonstrates server side validation. It is also possible to validate MVC applications by using client side validation. Client side validation will be covered in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

## Using Validation Helpers

When you request information from users, you often want the users to enter the data in a specific format so that it can be used further in the web application. For example, you might want to ensure that users enter a value for the **Name** property. You might also want to ensure that users enter a valid email address for the **EmailAddress** property. You can set these requirements by using validation data annotations in the model class. In controller actions, you can check the **ModelState.IsValid** property to verify if a user has entered valid data.

### • **Html.ValidationSummary()**

```
@Html.ValidationSummary()

 Please enter a name.
 The EmailAddress field is required

```

### • **Html.ValidationMessageFor()**

```
@Html.ValidationMessageFor(model => model.Name)

Please enter a name.
```

When users submit a form with invalid data, most websites display validation messages. These messages are often highlighted in red, but other colors or formats can be used. Often, there is a validation summary at the top of the page such as "Please enter valid information for the following fields: Last Name, Email Address". The form may display detailed validation messages next to each control in which the user entered invalid data, or highlight the problems in user-supplied data with an asterisk.

When your MVC view displays a form, you can easily show validation messages by using the validation helpers. Validation helpers render HTML only when users have entered invalid data.

### The **Html.ValidationSummary** HTML Helper

Use the **Html.ValidationSummary** HTML helper to render a summary of all the invalid data in the form. This helper is usually called at the top of the form. When the user submits invalid data, the validation messages are displayed for each invalid field in a bulleted list.

The following code example shows how you can change the **GetDetails** view to use the **Html.ValidationSummary** helper to render a summary of validation messages:

### Using the **Html.ValidationSummary** helper

```
@model ModelNamespace.Person

<h1>Get person details</h1>
@using (Html.BeginForm())
{
 @Html.ValidationSummary()

 @Html.EditorForModel()
 <input type="submit" value="Submit my details" />
}
```

### The **Html.ValidationMessageFor** HTML Helper

Use the **Html.ValidationMessageFor** HTML helper to render validation messages next to each input in the form.

The following code example shows how the **GetDetails** view can be changed to use the **Html.ValidationMessageFor** helper to render a validation message for the properties of the **Person** model class:

### Using the **Html.ValidationMessageFor** helper

```
@model ModelNamespace.Person

<h1>Get person details</h1>
@using (Html.BeginForm())
{
 @Html.ValidationSummary()

 @Html.LabelFor(p => p.Name)
 @Html.EditorFor(p => p.Name)
 @Html.ValidationMessageFor(p => p.Name)

 @Html.LabelFor(p => p.Age)
 @Html.EditorFor(p => p.Age)
 @Html.ValidationMessageFor(p => p.Age)

 @Html.LabelFor(p => p.EmailAddress)
 @Html.EditorFor(p => p.EmailAddress)
 @Html.ValidationMessageFor(p => p.EmailAddress)

 @Html.LabelFor(p => p.Description)
 @Html.EditorFor(p => p.Description)
 @Html.ValidationMessageFor(p => p.Description)

 <input type="submit" value="Submit my details" />
}
```

### The **ValidationMessageTagHelper** and **ValidationSummaryTagHelper** Tag Helpers

The **ValidationMessageTagHelper** tag helper is an alternative to the **Html.ValidationMessageFor** HTML helper. You can use the **ValidationMessageTagHelper** tag helper to display a validation message for a specific property of a model. You can use this tag helper by adding an **asp-validation-for** attribute to a **<span>** element.

The **ValidationSummaryTagHelper** tag helper is an alternative to the **Html.ValidationSummary** HTML helper that you can use to display validation messages that apply to the entire model. You can use this tag helper by adding an **asp-validation-summary** attribute to a **<div>** element. The value of the attribute **asp-validation-summary** attribute can be one of the following values:

- **All**. Using this value will cause both property and model level validation messages to be displayed.
- **ModelOnly**. Using this value will cause only model level validation messages to be displayed (excludes all property errors).
- **None**. Using this value will cause no validation summary to be displayed.

The following code example shows how you can change the **GetDetails** view to use tag helpers instead of using HTML helpers:

### Using the ValidationMessageTagHelper and ValidationSummaryTagHelper helpers

```
@model ModelNamespace.Person

<h1>Get person details</h1>
<form asp-controller="Person" asp-action="GetDetails">
 <div asp-validation-summary="All"></div>

 <label asp-for="Name"></label>
 <input asp-for="Name" />

 <label asp-for="Age"></label>
 <input asp-for="Age" />

 <label asp-for="EmailAddress"></label>
 <input asp-for="EmailAddress" />

 <label asp-for="Description"></label>
 <input asp-for="Description" />

 <input type="submit" value="Submit my details" />
</form>
```

## Demonstration: How to Validate User Input with Data Annotations

In this demonstration, you will first learn how to add validation data annotations to a model. Then, you will learn how to use the **ModelState.IsValid** property in the **HomeController** class. Finally, you will learn how to build a form in the view and add to the form validation helpers.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Validate User Input with Data Annotations" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_DEMO.md#demonstration-how-to-validate-user-input-with-data-annotations](https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D_MOD06_DEMO.md#demonstration-how-to-validate-user-input-with-data-annotations)

## Adding Custom Validations

You can use data annotations to indicate to MVC how it should validate the data that a user enters in a form or passes in query strings. Although the built-in validation attributes are very flexible, in some situations, such as the following examples, you might want to run some custom validation code:

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

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

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

### Creating custom validation data annotation

```
public class AllLettersValidationAttribute : ValidationAttribute
{
 public override bool IsValid(Object value)
 {
 return ((string)value).All(Char.IsLetter);
 }
}
```

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

### Using a custom validation data annotation

```
public class Person
{
 [Display(Name = "My Name")]
 [Required(ErrorMessage = "Please enter a name.")]
 [AllLettersValidation(ErrorMessage = "Only letters allowed.")]
 public string Name { get; set; }

 [Range(0, 150)]
 public int Age { get; set; }

 [Required]
 [RegularExpression(".+\\@.+\\..+")]
 public string EmailAddress { get; set; }

 [DataType(DataType.MultilineText)]
 [StringLength(20)]
 public string Description { get; set; }
}
```

#### • Create custom validation data annotations

```
public class AllLettersValidationAttribute : ValidationAttribute
{
 public override bool IsValid(Object value)
 {
 return ((string)value).All(Char.IsLetter);
 }
}
```

#### • Use custom validation data annotations

```
[AllLettersValidation(ErrorMessage = "Only letters allowed.")]
public string Name { get; set; }
```

Notice that the **IsValid** method of the **ValidationAttribute** class is overloaded. In case you need an access to the whole model class, you can override another version of the **IsValid** method, which gets as a second parameter an object of type **ValidationContext**.

The following lines of code illustrate how you can access the model class in the **IsValid** method:

#### Accessing the Model class in the **IsValid** method

```
public class AllLettersValidationAttribute : ValidationAttribute
{
 protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
 {
 Person person = (Person)validationContext.ObjectInstance;
 if (!person.Name.All(Char.IsLetter))
 {
 return new ValidationResult("All characters in Name must be letters");
 }
 return ValidationResult.Success;
 }
}
```

You can use the *ValidationContext* parameter to use a service that was registered in the **ConfigureServices** method of the **Startup** class.

 **Note:** Services were covered in Module 3, "Configure Middleware and Services in ASP.NET Core".

The following lines of code illustrate how you can use a service in the **IsValid** method:

#### Using a service in the **IsValid** method

```
public class AllLettersValidationAttribute : ValidationAttribute
{
 protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
 {
 var service = (IMyService)validationContext.GetService(typeof(IMyService));
 Person person = (Person)validationContext.ObjectInstance;
 if (!person.Name.All(Char.IsLetter))
 {
 return new ValidationResult("All characters in Name must be letters");
 }
 return ValidationResult.Success;
 }
}
```

## Demonstration: How to Add Custom Validations

In this demonstration, you will learn how to add custom validations.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Add Custom Validations" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_DEMO.md#demonstration-how-to-add-custom-validations](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD06_DEMO.md#demonstration-how-to-add-custom-validations).

# Lab: Developing Models

## Scenario

You are planning to create and code a Model-View-Controller (MVC) model that will be used in the butterflies' shop application. The model includes properties that describe a butterfly. The model must enable the application to store the uploaded butterflies.

## Objectives

After completing this lab, you will be able to:

- Add new models to the application, and add properties to the model.
- Add **GET** and **POST** actions that accept the new model information.
- Use display and edit data annotations in the MVC model to assign property attributes to views and controllers.
- Use **Display**, **Editor** and **Form** Helpers inside the views.
- Use validation data annotations in the MVC model to assign property attributes to views and controllers.
- Add custom validation to the application.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD06_LAB_MANUAL.md)

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD06\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD06_LAK.md)

## Exercise 1: Adding a Model

### Scenario

In this exercise, you will:

- Add models to the web application and use them in views.
- Add a **Create (GET)** action.
- Add a **Create (POST)** action.

The main tasks for this exercise are as follows:

1. Create a new model.
2. Use the model in a view.
3. Pass the model from the controller to a view.
4. Run the application.
5. Write a GET action.
6. Write a POST action that accepts the model.

## Exercise 2: Working with Forms

### Scenario

In this exercise, you will:

- Add Display and Edit data annotations to a **Butterfly** model.
- Add **Display** Helpers to **Index.cshtml**.
- Add **Form** Helpers and **Editor** Helpers to **Create.cshtml**.

The main tasks for this exercise are as follows:

1. Add display and edit data annotations to a model.
2. Update an action to return FileContentResult.
3. Add Display Helpers.
4. Add Form Helpers.
5. Add Editor Helpers.
6. Run the application.

## Exercise 3: Adding Validation

### Scenario

In this exercise, you will:

- Add validation data annotations to a butterfly model.
- Add validation Helpers to the Create view.
- Add **ModelState.IsValid** property in ButterflyController.
- Add **MaxButterflyQuantityValidation** custom validation.

The main tasks for this exercise are as follows:

1. Add validation data annotations to a model.
2. Add validation helpers to a view.
3. Using **ModelState.IsValid** property in a controller.
4. Run the application.
5. Add custom validation.
6. Run the application.

**Question:** In your web application, you want users to be able to enter their email address in addition to their butterfly information. What should you do so that the form displays a text box, which will allow the users to type their email address?

**Question:** If a user does not fill the butterfly information in the form and clicks the submit button, what will happen?

# Module Review and Takeaways

The model class is an important part of an MVC web application because it describes the information and objects that your web application manages. In this module, you saw how to create your model, and how to display, edit, and validate properties. You examined how to use HTML helpers and tag helpers that are built into ASP.NET Core MVC to facilitate the rendering of displays, controls, and forms, and return validation messages to users.

## Review Question

**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?

## Best Practice

You should put the models of an ASP.NET Core MVC application in a folder named **Models**.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The browser doesn't display correctly the tag helpers of your form.	



# Module 7

## Using Entity Framework Core in ASP.NET Core

### Contents:

Module Overview	07-1
<b>Lesson 1:</b> Introduction to Entity Framework Core	07-2
<b>Lesson 2:</b> Working with Entity Framework Core	07-10
<b>Lesson 3:</b> Using Entity Framework Core to Connect to Microsoft SQL Server	07-22
<b>Lab:</b> Using Entity Framework Core in ASP.NET Core	07-35
Module Review and Takeaways	07-37

## Module Overview

Web applications often use information and they usually require a data store for that information. By rendering webpages that use data from a data store, you can create a web application that changes continually in response to user input, administrative actions, and publishing events. The data store is usually a database, but other types of data stores are occasionally used. In Model-View-Controller (MVC) applications, you can create a model that implements data access logic and business logic. Alternatively, you can separate business logic from data access logic by using a repository. A repository is a class that a controller can call to read data from a data store and to write data to a data store. When you write an ASP.NET application you can use the Entity Framework Core (EF Core) and Language Integrated Query (LINQ) technologies, which make data access code very quick to write and simple to understand. In this module, you will see how to build a database-driven website in MVC.

### Objectives

After completing this module, you will be able to:

- Connect an application to a database to access and store data.
- Explain EF Core.
- Work with Entity Framework Core.
- Use EF Core to connect to a database including Microsoft SQL Server.

## Lesson 1

# Introduction to Entity Framework Core

EF Core is an object-relational mapping (ORM) framework that can be used when developing ASP.NET Core applications. It allows developers to work with databases and use the data in ASP.NET Core applications. Moreover, it enables developers to be data-oriented without the need to concentrate on modeling the entities. EF Core is a light-weight version of Entity Framework. In addition, it is also extensible and cross-platform.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe what is ADO.NET and its uses.
- Explain what is ORM.
- Explain the advantages of ORM.
- Describe what is Entity Framework.
- Explain the differences between Entity Framework 6 and EF Core.
- Explain what is a database provider.
- Add a database provider to your application.

## Connecting to a Database Using ADO.NET

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

- ADO.NET is a basic data access API that contains a set of data providers
- Data providers connect to various databases
- ADO.NET providers consist of:
  - **Connection**. Manages a connection to a database.
  - **Command**. Represents a query or manipulation operation.
  - **DataReader**. Forward-only, cursor interface for queries.
  - **DataAdapter**. Tabular interface for queries.

### ADO.NET and Databases

When you create .NET Framework applications, including ASP.NET Core MVC web applications, you can use the ADO.NET technology to access databases. ADO.NET is a basic data access API in the .NET Framework and contains a set of *data providers* that support most of the free and commercial databases available today. A data provider implements database-specific protocols and features and at the same time presents a consistent API so that replacing the application's data provider does not involve many code changes.

ADO.NET contains several data providers, which include:

- **System.Data.SqlClient**. Used to connect to SQL Server databases and Microsoft Azure SQL databases.
- **System.Data.Odbc**. Used to connect to databases that support the ODBC API.

You can also find third-party data providers online and you can implement your own data provider.

## The ADO.NET Connection Object

You can use the ADO.NET *connection* object to connect to a database. Each provider has an ADO.NET connection object that implements the **IDbConnection** interface:

- **SqlConnection**
- **OdbcConnection**

A connection object connects to the database and initiates additional operations, such as executing commands or managing transactions. Typically, you create a connection object with a *connection string*, which is a locator for your database and might contain connection-related settings, such as authentication credentials and timeout settings.

The following code example demonstrates how to connect to a SQL Express database called **PhotoSharingDB** by using the credentials under which the web application runs:

### Connecting to SQL Express

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 string connectionString = "Data Source=.\SQLEXPRESS;Initial Catalog=PhotoSharingDB;" +
 "Integrated Security=SSPI";
 using (SqlConnection conn = new SqlConnection(connectionString))
 {
 conn.Open();
 }

 return View();
 }
}
```

The following code example demonstrates how to connect to a Microsoft Azure SQL database:

### Connecting to a Microsoft Azure SQL Database

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 string connectionString =
 "Server=tcp:example.database.windows.net,1433;Database=PhotoSharingDB;" +
 "User ID=Admin@example;Password=Pa$$w0rd;Trusted_Connection=False;" +
 "Encrypt=True;Connection Timeout=30;PersistSecurityInfo=true";
 using (SqlConnection conn = new SqlConnection(connectionString))
 {
 conn.Open();
 }

 return View();
 }
}
```



**Note:** Microsoft Azure will be covered in Module 14, “Hosting and Deployment”.

## The ADO.NET Command Object

You can use the ADO.NET **command** object to send commands to a database. Commands can return data, such as the result of a **select** query or a stored procedure. Alternately, commands may not return any data, such as when you use an **insert** or **delete** statement or a Data Definition Language (DDL) query. Each provider has an ADO.NET command object that implements the **IDbCommand** interface:

- **SqlCommand**
- **OdbcCommand**

A command object can represent a single command or a set of commands. Query commands return a set of results as a **DataReader** object or a **DataSet** object or a single value, usually, the result of an aggregated action, such as a row count, or calculation of an average.

## The ADO.NET DataReader Object

You can use the ADO.NET data reader object to dynamically iterate a result set obtained from a database. If you use a data reader to access data, you must maintain a live connection while you read from the database. Additionally, data readers can only move forward while iterating the data. This data-access strategy is also referred to as the *connected architecture*. Each provider has an ADO.NET data reader object that implements the **IDataReader** interface:

- **SqlDataReader**
- **OdbcDataReader**

The following code example demonstrates how to query a database with a data reader in a controller:

### Querying a Database with a Data Reader

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 List<string> cities = new List<string>();
 string connectionString = "Server=(localdb)\\sqlexpress;Database=MyDB;Integrated
Security=True;";
 using (SqlConnection conn = new SqlConnection(connectionString))
 {
 conn.Open();

 using (SqlCommand command = new SqlCommand("select * from city", conn))
 {
 using (SqlDataReader reader = command.ExecuteReader())
 {
 while (reader.Read())
 {
 string city = (string)reader["Name"];
 cities.Add(city);
 }
 }
 }

 return View(cities);
 }
 }
}
```

When you use a data reader, you can access only one database record at a time, as shown in the preceding example. If you need multiple records at once, you must store them as you move to the next record. Although this seems like a major inconvenience, data readers are very efficient in terms of memory utilization because they do not require the entire result set to be fetched into memory.

## The ADO.NET DataAdapter Object

You can use the ADO.NET *data adapter* object to load a result set obtained from a database into the memory. After loading the entire result set and caching it in the memory, you can access any of its rows, unlike the data reader, which only provides forward iteration. You should use this data-access strategy, referred to as the *disconnected architecture*, when you do not want to maintain a live connection to the database while processing the data.

Data adapters store the results in a tabular format. You can also change the data after it is loaded and use the data adapter to apply the changes back to the database. Each provider has an ADO.NET data adapter object that implements the **IDataAdapter** interface:

- **SqlDataAdapter**
- **OdbcDataAdapter**

Although data adapters are convenient to use (especially in conjunction with the **DataSet** class, which is explained in the next section), they impose a larger overhead than data readers because the entire result set must be fetched into memory before you can perform any operations.

## The DataSet Object

The **DataSet** object is one of the most frequently used objects in ADO.NET. You use it to retrieve tabular data from a database. Although you can fill a **DataSet** object manually with data, you typically load it by using the **DataAdapter** class.

The following code example demonstrates how to load data to a **DataSet** object by using a data adapter in a controller:

### Loading Data into a DataSet Object with the SqlDataAdapter Class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 List<string> cities = new List<string>();
 string connectionString = "Server=(localdb)\\sqlexpress;Database=MyDB;Integrated Security=True;";
 string query = "select * from city";
 using (SqlDataAdapter adapter = new SqlDataAdapter(query, connectionString))
 {
 DataSet dataSet = new DataSet();
 adapter.Fill(dataSet);
 foreach (DataRow row in dataSet.Tables["Table"].Rows)
 {
 string city = (string)row["name"];
 cities.Add(city);
 }
 }

 return View(cities);
 }
}
```

You can use the **DataSet** objects to hold information from more than one table at one time and to maintain relationships between the tables.

## Object Relational Mapper (ORM)

Developers write code that works with classes and objects. In contrast, databases store data in tables with columns and rows, and database administrators create and analyze databases by running Transact-SQL queries.

When you use ADO.NET to interact with your database, your application becomes strongly linked to the database. With ADO.NET, you implement most of your data access as plain-text SQL statements or you call stored procedures implemented on the database server in some SQL dialect. This approach is error-prone and inflexible. Changing the database schema might require considerable modifications to the code of the application.

ORM is a programming technique that simplifies the application's interaction with data by using a metadata descriptor to connect object code to a relational database. ORM provides an abstraction that maps application objects to database records.

There are multiple ORM frameworks. You can choose the one which is appropriate for you. One of the most popular ORM frameworks is Entity Framework. Entity Framework was created by Microsoft and it maps the tables and columns found in a database to the objects and properties that are used in .NET code.

Besides Entity Framework, there are also other ORM frameworks such as Hibernate and Django. Nevertheless, Entity Framework was made for .NET Framework by Microsoft as the best match for .NET applications. The EF Core is a version which can run on different operating systems and work with different databases.

- ORM is an approach designed to simplify the interaction with data
- There are multiple ORM frameworks
- Entity Framework is an ORM framework that was created for .NET
- ORM maps the tabular structure into data model classes
- You can use an ORM framework to modify objects in a database

## Overview of Entity Framework

Entity Framework provides a one-stop solution to interact with data that is stored in a database. Instead of writing plain-text SQL statements, you can work with your own domain classes, and you do not have to parse the results from a tabular structure to an object structure. These domain classes are called *entities*. Entity Framework keeps track of the changes you make to the entities to enable updating the database with data that exists in memory.

Entity Framework introduces an abstraction layer between the database schema and the code of your application, which makes your application more flexible. Using Entity Framework, you can access objects by using strongly typed code and query them by using LINQ. Therefore, you no longer have to rely on SQL statements. This makes your code more robust.

- Entity Framework provides a one-stop solution to interact with data that is stored in a database
- Entity Framework Approaches:
  - Database First
  - Model First
  - Code First
- Entity Framework Versions:
  - Entity Framework 6 (EF6)
  - Entity Framework Core (EF Core)

## Entity Framework Approaches

Entity Framework provides three general approaches to create your data access layer (DAL) of the application:

- Database First
- Model First
- Code First

In each of these approaches, Entity Framework can either create a new database, according to the data model or use an existing database.

### **Model First and Database First**

In these approaches, the data model is generated by using a designer in Visual Studio. The model is stored in a .edmx file. The entity classes and the relationships are generated from the model.

If you do not already have a database, after you design your data model, you can generate database scripts from the model by using the Visual Studio designer. You can then run the script on a new database to create the tables. On the other hand, if your database existed prior to creating the data model, you can use the Visual Studio designer to reverse engineer the data model from the database tables.

### **Code First**

In this approach, you do not use a .edmx file to design your model, and you do not rely on the Visual Studio designer. The domain model is simply a set of classes with properties that you provide. In the Code First approach, Entity Framework scans your domain classes and their properties and tries to map them to the database.

You can use the Code First approach both with new databases and with existing ones. If you do not have a database, the default behavior of the Code First approach is to create the database the first time you run your application. If your database already exists, Entity Framework will connect to it and use the defined mappings between your model classes and the existing database tables.

## Entity Framework Versions

There are different versions of Entity Framework and it is important to be familiar with them.

### **Entity Framework 6 (EF6)**

Entity Framework 6 (EF6) was first released in 2008. With EF6 you can control the data stored in the database and use it within your application. It supports the Model First, Database First, and Code First approaches, which allow you to choose the design approach for your application. However, EF6 is not supported on operating systems other than Windows.

### **Entity Framework Core (EF Core)**

After the decision of making .NET Framework *cross platform*, supporting several operating systems and not only Windows, there was a need to re-implement Entity Framework. As a result, EF Core was created. EF Core is the .NET Core version of Entity Framework. The EF Core is extensible; hence it comes lightweight giving you the minimum needed features to work with and is capable of adding more features by request. Moreover, EF Core is open source, hence you can find solutions or specific features as well as creating your own. It is important to note that EF Core works only with the Code First approach. However, it is a modern, advanced and rising framework that can be used widely and by a great variety of platforms.

In this course, you will learn how to use EF Core in ASP.NET Core applications.

## Discussion: Choose between Entity Framework Core and Entity Framework 6

The following scenarios describe some requirements for a website. In each case, discuss which version of Entity Framework you would choose to implement the required functionality.

### Working with a non-windows environment

Your company is developing a product and work with a Linux environment. It wants to connect the application to an Oracle database. You are asked to create an object-oriented model and connect it to that database.

Which Entity Framework version will you use in the following scenarios? (Choose one of the following in each scenario: Entity Framework 6, Entity Framework Core)

- Working with a non-Windows environment
- Developing by using the Database First approach
- Creating a UWP application with a NoSql database

### Developing by using the Database First approach

The architect of your company designed a database alongside with the DB Leader and now asked you to create an application based on that database. You are also asked to work with Database First approach when you are developing the application.

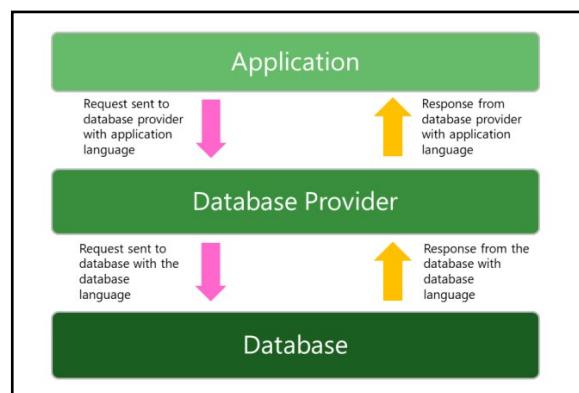
### Creating a Universal Windows Platform (UWP) application with a NoSql database

You have a new assignment. This assignment requires to create a UWP application so the customers of that application can use it on different devices. Your application should be able to store data in a NoSql database.

## Database Providers

EF Core is a layer between your code and a database. It is used to connect your code to the database. A database provider is a library which is used by EF Core to connect to a specific database. Database providers are distributed as NuGet packages. If you want to use a database provider in your application, you can install the corresponding NuGet package.

Before you choose a database provider, it is important that you know the requirements of your application. For instance, the type of the database you use, whether you are willing to pay for the database provider, and the type of your development environment. All these requirements are crucial for the decision of which database provider you should use.



### The Microsoft SQL Provider

A very commonly used database provider in EF Core is the SQL Server provider. The SQL Server provider enables you to connect a SQL Server database to your application by using EF Core. SQL Server database is a Microsoft's relational database with a tabular structure.



**Note:** Connecting to a SQL Server database by using the Microsoft SQL Provider will be covered in Lesson 3, "Using Entity Framework Core to Connect to Microsoft SQL Server".

## The SQLite Provider

SQLite is a popular, open source database that is widely used by developers. You can use the SQLite provider to connect an SQLite database to your application by using EF Core.

 **Note:** Connecting to an SQLite database by using the SQLite provider will be covered in Lesson 2, "Working with Entity Framework Core".

## The InMemory Provider

Entity Framework Core introduced the InMemory provider. It is important to point out the InMemory provider does not connect to a real relational database and does not mimic a database, but it is designed to be a tool for testing your application. For example, with InMemory provider you are able to save data that you will not be able to save in a real relational database such as data that may violate referential integrity constraints.

## Other Providers

There are plenty of other database providers. Using these providers, it is possible to connect to various database engines such as MySQL, Maria DB and Db2. However, notice that while some database providers are maintained by the EF Core Project (Microsoft) vendor, others database providers might be maintained by other vendors. Before using a database provider, ensure that it meets your requirement.

## Adding a Database Provider to Your Application

Usually a database provider is distributed as a NuGet package. Therefore, it is quite simple to add a database provider to your application. To add a database provider to your application you can either use the dotnet tool or use the NuGet Package Manager Console.

 **Note:** The dotnet tool will be covered in Module 14, "Hosting and Deployment".

This is the dotnet command you can use to add a database provider:

### Add a Database Provider by using the dotnet Tool

```
dotnet add package provider_package_name
```

This is the NuGet Package Manager Console command you can use to add a database provider:

### Adding a Database Provider by using a NuGet Package Manager Console Command

```
install-package provider_package_name
```

## Lesson 2

# Working with Entity Framework Core

You can use EF Core in your ASP.NET Core applications to connect to a database. To connect to a database by using EF Core you can use an Entity Framework context. An Entity Framework context is a class that inherits from the **DbContext** class. In this lesson, you will learn how to configure an Entity Framework context to connect to a SQLite database.

The Entity Framework context can be used to retrieve data from a database. To retrieve data from a database by using an Entity Framework context, you can use LINQ to Entities. Furthermore, you can also load related data by using EF Core. In this lesson, you will see several techniques to load related data by using EF Core, including the following ORM patterns: explicit loading, eager loading, and lazy loading.

You can also use the Entity Framework context to manipulate data in the database. In this lesson, you will see how to use the Entity Framework context to add new records to a database, update an existing record in a database, and delete a record from a database.

## Lesson Objectives

After completing this lesson, you will be able to:

- Create an entity.
- Explain what is an Entity Framework context.
- Connect EF Core to a SQLite database.
- Retrieve data from a database by using EF Core.
- Explain the different techniques of loading related data.
- Manipulate data in a database by using EF Core.

## Using an Entity Framework Context

In EF Core, a *context* is how you access the database, without the need for additional wrappers or abstractions. An Entity Framework context is the glue between your domain model classes and the underlying framework that connects to the database and maps object operations to database commands.

An Entity Framework context is used to:

- Provide basic create, read, update, and delete (CRUD) operations, and to simplify the code that you need to write to execute these operations.
- Handle the opening and closing of database connections.
- Handle the database generation when working with the Code First approach.

• **An entity:**

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
}
```

• **Entity Framework context:**

```
public class HrContext : DbContext
{
 public HrContext(DbContextOptions<HrContext> options) : base(options)
 {
 }

 public DbSet<Person> Candidates { get; set; }
}
```

## Adding an Entity Framework Context

To coordinate with Entity Framework functionalities, you will have to create an Entity Framework context. A context in Entity Framework Core is a class that inherits from the **DbContext** class. You can use this class when you want to access the database.

The following example demonstrates how to create a custom class that derives from the **DbContext** class:

### A Class Deriving from the DbContext Class

```
public class HrContext : DbContext
{
}
```

## Domain Model Classes (Entities)

In addition to creating an Entity Framework context, you should also create the domain model classes of your application. Domain model classes are “plain-old CLR objects” (POCO) classes. The reason for this name is that they do not have any dependency on EF Core. They just define the properties of the data that is stored in the database. The domain model classes are called *entities*. In your code, you specify which entities are included in the data model and you can also customize certain Entity Framework behavior for each entity.

The following code shows an entity class:

### An Entity Class

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
}
```

## Adding DbSet Properties to an Entity Framework Context

When working with EF Core, you want to ensure that the entities defined in the code are mapped to tables defined in the database. To achieve this, you can add the **DbSet<>** properties to the Entity Framework context. Each **DbSet<>** property gets an entity type as a type parameter. In relational databases, the entity is mapped to a table in the database. EF Core will map the properties according to the mapping information provided by the classes.

The following code example demonstrates how to add a **DbSet<Person>** property to the **HrContext** class:

### Adding a DbSet<> property to an Entity Framework Context

```
public class HrContext : DbContext
{
 public DbSet<Person> Candidates{ get; set; }
}
```

## Connecting an Entity Framework Context to SQLite Database

You can connect an Entity Framework context to a database. To connect an Entity Framework context to a database, you need to use a database provider which corresponds to the database chosen.

 **Note:** Database providers were covered in Lesson 1, “Introduction to Entity Framework Core”.

One of the most popular databases that you can access when using EF Core is SQLite. SQLite is a serverless database, which means that you don't need to have a separate server to use it. One of the big advantages of SQLite, when compared to other databases, is that it is very easy to configure.

You can use the SQLite database to store data in a file. In this lesson, Entity Framework will be configured to connect to a SQLite database, and the data in the database will be stored in a file on disk.

 **Note:** Configuring EF Core to connect to a SQL Server will be covered in Lesson 3, "Using Entity Framework Core to Connect to Microsoft SQL Server".

To use an Entity Framework context in an ASP.NET Core application, you need to register a service by using the **AddDbContext<>** method, passing it the Entity Framework context class type as a generic type parameter. You should register the service in the **ConfigureServices** method of the **Startup** class.

The following code shows how to configure HrContext to connect to a SQLite database which will be stored in a file named example.db:

#### Configuring HrContext to connect to a SQLite Database

```
public class Startup
{
 public void ConfigureServices(IServiceCollection services)
 {
 services.AddDbContext<HrContext>(options => options.UseSqlite("Data
Source=example.db"));
 services.AddMvc();
 }

 public void Configure(IApplicationBuilder app)
 {
 app.UseMvc();
 }
}
```

 **Note:** The SQLite database provider is distributed with the Microsoft.EntityFrameworkCore.Sqlite NuGet package.

The following example demonstrates how the Entity Framework context can receive the options specified in the **ConfigureServices** method:

#### The HrContext Class

```
public class HrContext : DbContext
{
 public HrContext(DbContextOptions<HrContext> options) : base(options)

 public DbSet<Person> Candidates { get; set; }
}
```

#### Using an Entity Framework Context in a Controller

After configuring the Entity Framework context, you can use it in a controller. The Entity Framework context is passed to the constructor of the controller you are using due to the dependency injection mechanism.

The following code shows a controller that gets a **HrContext** instance by using the dependency injection mechanism:

### Using Dependency Injection

```
public class HrController : Controller
{
 private HrContext _context;

 public HrController(HrContext context)
 {
 _context = context;
 }

 public IActionResult Index()
 {
 return View(_context.Candidates.ToList());
 }
}
```

### Creating and Deleting a Database

You can use the **Database** property of a **DbContext** object to create a database or delete a database. By using the **Database** property of the **DbContext** object, you can call the **EnsureCreated** and **EnsureDeleted** methods. When you call the **EnsureCreated** method, Entity Framework creates a database based on the information in your **DbContext**-derived class if the database does not already exist. When you call the **EnsureDeleted** method, Entity Framework deletes the database if it already exists.

The following code shows how to delete a database if it exists by using the **EnsureDeleted** method, and how to create a new database by using the **EnsureCreated** method:

### Using the EnsureDeleted and EnsureCreated Methods

```
public void Configure(IApplicationBuilder app, HrContext ctx)
{
 ctx.Database.EnsureDeleted();
 ctx.Database.EnsureCreated();

 app.UseMvc();
}
```

### Data Seeding

You can use a data seeding to populate the database with sample data when a database is created. When you call the **EnsureCreated** method, a new database will be created and initialized with the sample data. Notice that in case a database already exists when **EnsureCreated** is called, the sample data won't be added to the database.

The following code demonstrates how to seed data by using the **OnModelCreating** method of the **HrContext** class:

### Seeding Data Code

```
public class HrContext : DbContext
{
 public HrContext(DbContextOptions<HrContext> options) : base(options)
 {}

 public DbSet<Person> Candidates { get; set; }

 protected override void OnModelCreating(ModelBuilder modelBuilder)
 {
```

```

 modelBuilder.Entity<Person>().HasData(
 new { PersonId = 1, FirstName = "James", LastName = "Smith" },
 new { PersonId = 2, FirstName = "Arthur", LastName = "Adams" }
);
 }
}

```

 **Note:** In addition to seeding data by using the **EnsureCreated** method, you can also seed data when creating a new database by using migrations. Migrations will be covered in Lesson 3, "Using Entity Framework Core to Connect to Microsoft SQL Server".

## Using LINQ to Entities

LINQ is a set of extension methods that enable you to write complex query expressions. You can use these expressions to extract data from databases, enumerable objects, XML documents, and other data sources. The expressions are similar to Transact-SQL queries, but you can get IntelliSense support and error checking in Visual Studio.

### What Is LINQ to Entities?

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

- LINQ to Entities is the version of LINQ that works with Entity Framework
- Sample LINQ Query:

```

var list = from c in _context.Candidates
 where c.LastName == "Smith"
 select c;

```

### The LINQ Query

The query is specifying the information which you want to get from your database. Moreover, you have the option to specify how the retrieved information will be sorted, grouped or formatted before it is returned. The LINQ query is stored in a query variable and initialized with a query expression. A query expression is a query expressed in query syntax, it is a first-class language construct and it can be used in any context in which a C# expression is valid. You should notice that the writing of LINQ query is slightly different than writing a SQL query.

The following example shows how to retrieve a list of people from the database and filter it by the last name of the person:

### Querying Data by using LINQ to Entities

```

public IActionResult Index()
{
 var list = from c in _context.Candidates
 where c.LastName == "Smith"
 select c;
 return View(list);
}

```

## Loading Related Data

In EF Core you can load related entities by using navigation properties. To load related data, you need to choose an ORM pattern. EF Core contains several ORM patterns, which include:

- **Explicit loading.** Using this pattern, the related data is loaded explicitly from the database after the original query is completed.
- **Eager loading.** Using this pattern, the related data is loaded from the database as part of the original query.
- **Lazy loading.** Using this pattern, the related data is loaded from the database as you access the navigation property.

- In Entity Framework Core you can load related entities by using navigation properties
- To load related data, you need choose an ORM pattern
- Entity Framework Core contains several ORM patterns, which include:
  - Explicit loading
  - Eager loading
  - Lazy loading

## Navigation Properties

In EF Core you can link an entity to other entities using by navigation properties. When an entity is related to another entity, you should add a navigation property to represent the association. When the multiplicity of the association is one or zero-or-one, the navigation property is represented by a reference object. When the multiplicity of the association is many, the navigation property is represented by a collection.

The following code example shows an entity named **Country**. Since the **Country** entity participates in a one-to-many relationship with the **City** entity, it contains a navigation property named **Cities** of type **List<City>**. Similarly, since the **Country** entity participates in a one-to-many relationship with the **Person** entity, it contains a navigation property named **People** of type **List<Person>**:

### An Entity named Country with Navigation Properties

```
public class Country
{
 public int CountryId { get; set; }
 public string Name { get; set; }
 public int Size { get; set; }
 public List<City> Cities { get; set; }
 public List<Person> People { get; set; }
}
```

The following code example shows an entity named **City**. Since the **City** entity participates in a one-to-many relationship with the **Person** entity, it contains a navigation property named **People** of type **List<Person>**. Since each city belongs to exactly one country, the **City** entity contains a navigation property named **Country**. The **CountryId** property represents a foreign key:

### An Entity named City with Navigation Properties

```
public class City
{
 public int CityId { get; set; }
 public string Name { get; set; }
 public int Size { get; set; }
 public int CountryId { get; set; }
 public Country Country { get; set; }
 public List<Person> People { get; set; }
}
```

The following code example shows an entity named **Person** with navigation properties to the **Country** and **City** entities:

### An Entity named Person with Navigation Properties

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public int CountryId { get; set; }
 public Country Country { get; set; }
 public int CityId { get; set; }
 public City City { get; set; }
}
```

The following example shows an Entity Framework context that contains collections of the entities:

### The DemographyContext Class

```
public class DemographyContext : DbContext
{
 public DemographyContext(DbContextOptions<DemographyContext> options) : base(options)
 {}

 public DbSet<Country> Countries { get; set; }
 public DbSet<City> Cities { get; set; }
 public DbSet<Person> People { get; set; }
}
```

### Explicit Loading

To load related entities by using the explicit loading ORM pattern, you should use the **Entry** method of the Entity Framework context class.

The following code example demonstrates how to load a city with its related entities by using explicit Loading:

### An Example of Explicit Loading

```
public class HomeController : Controller
{
 private DemographyContext _context;

 public HomeController(DemographyContext context)
 {
 _context = context;
 }

 public IActionResult Index()
 {
 var city = _context.Cities
 .Single(c => c.CityId == 1);

 _context.Entry(city)
 .Collection(c => c.People)
 .Load();

 _context.Entry(city)
 .Reference(c => c.Country)
 .Load();

 return View(city);
 }
}
```

```
}
```

In the preceding code, when the **city** variable is initialized, its **People** property is null. Only after the first call to the **Load** method is completed does the **People** property contain a list of people who live in the city. Similarly, when the **city** variable is initialized, its **Country** property is null. Only after the second call to the **Load** method is completed does the **Country** property contain the country to which the city belongs.

You can also use the explicit loading ORM pattern in conjunction with LINQ. To do so, you should first call the **Query** method, and then you can call the LINQ methods such as **Count** and **Where**.

The following code example demonstrates how to retrieve the number of people that live in a city:

### Explicit Loading with LINQ

```
var city = _context.Cities
 .Single(c => c.CityId == 1);

int num = _context.Entry(city)
 .Collection(c => c.People)
 .Query()
 .Count();
```

### Eager Loading

In addition to loading related data by using the explicit loading ORM pattern, you can also choose to load related data by using the eager loading ORM pattern. However, while loading related data by using the explicit loading ORM pattern is executed explicitly after the original query is completed, when the eager loading ORM pattern is used the related data is loaded as part of the original query.

To load related entities by using the explicit loading ORM pattern you need to use the **Include** method. The **Include** method specifies related entities to be included in the query results.

The following code example demonstrates how you can use eager loading to load cities with people that live in them from the database. Since the **Include** method is used, the **People** property of each city in the **cities** variable contains the people which live in the city:

### An example of Eager Loading

```
var cities = _context.Cities
 .Include(c => c.People)
 .ToList();
```

In case you need to include related data from multiple relationships, you can use the **Include** method several times in the same query.

The following code example demonstrates how to load cities with related entities from the database. The **People** property for each city in the **cities** variable contains the people that live in the city, and the **Country** property of each city in the **cities** variable contains the country to which the city belongs:

### Loading Related Data from Multiple Relationships

```
var cities = _context.Cities
 .Include(c => c.People)
 .Include(c => c.Country)
 .ToList();
```

If you need to include more levels of related data, you can use the **ThenInclude** method. The **ThenInclude** method can be used to drill down through the relationships.

The following code example demonstrates how to load all countries, their related cities, and the people that live in each city:

### Using the ThenInclude Method

```
var countries = _context.Countries
 .Include(country => country.Cities)
 .ThenInclude(city => city.People)
 .ToList();
```

### Lazy Loading

The lazy loading ORM pattern can be used when you want to load related data from the database as you access the navigation property.

In case you want that EF Core will use lazy loading to load a navigation property, you should change the navigation property to be overridden. To change a navigation property to be overridden you can use the `virtual` keyword.

The following code example demonstrates how the **Country**, **City** and **People** entities can be changed so their navigation properties will be overridden:

### Entities with Lazy Loaded Navigation Properties

```
public class Country
{
 public int CountryId { get; set; }
 public string Name { get; set; }
 public int Size { get; set; }
 public virtual ICollection<City> Cities { get; set; }
 public virtual ICollection<Person> People { get; set; }
}

public class City
{
 public int CityId { get; set; }
 public string Name { get; set; }
 public int Size { get; set; }
 public int CountryId { get; set; }
 public virtual Country Country { get; set; }
 public virtual ICollection<Person> People { get; set; }
}

public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public int CountryId { get; set; }
 public virtual Country Country { get; set; }
 public int CityId { get; set; }
 public virtual City City { get; set; }
}
```

If you want to use the lazy loading ORM pattern, in addition to changing the navigation properties to be overridden, you should also turn on the creation of lazy-loading proxies. To turn on the creation of lazy-loading proxies, you can call the **UseLazyLoadingProxies** method.

The following code example demonstrates how to turn on the creation of lazy-loading proxies for the **DemographyContext** class:

## Calling the `UseLazyLoadingProxies` Method

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddDbContext<DemographyContext>(
 options => options.UseLazyLoadingProxies()
 .UseSqlite("Data Source=example.db"));
 services.AddMvc();
}
```

 **Note:** The `UseLazyLoadingProxies` method is distributed with the `Microsoft.EntityFrameworkCore.Proxies` NuGet package.

The following code example demonstrates how you can retrieve the number of people that live in a city, and to which country the city belongs:

### An Example of Lazy Loading

```
City city = _context.Cities.First(c => c.CityId == 1);
int numOfPeople = city.People.Count;
Country country = city.Country;
```

## Manipulating Data by Using Entity Framework

EF Core can track entities that you retrieve from the database. EF Core uses change tracking so that when you call the `SaveChanges` method on the Entity Framework context object, it can synchronize your updates with the database. You can check the status of any entity (such as whether it was modified), inspect the history of your changes, and undo changes if required.

Each entity in EF Core can be in one of the following states:

- **Added.** The entity was added to the context and does not exist in the database.
- **Modified.** The entity was changed since it was retrieved from the database.
- **Unchanged.** The entity was not changed since it was retrieved from the database.
- **Deleted.** The entity was deleted since it was retrieved from the database.

- Entity Framework can track your entity changes
- The context uses in-memory snapshots to detect changes
- Call the `SaveChanges` method to save changes to the database

```
_context.People.Remove(person);
_context.SaveChanges();
```

### Inserting New Entities

To add an entity to the database, you can use the Entity Framework context object. When you use the Entity Framework context object to add a new entity to a database, the context marks the change tracking status of the entity as **Added**. When you call the `SaveChanges` method, the Entity Framework context object adds the entity to the database. No changes are applied to the database until you call the `SaveChanges` method.

The following code example shows an entity named **Person**:

### An Entity

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
}
```

The following code example shows how to add a new entity to a database by using the Entity Framework context object:

### Adding an Entity

```
public IActionResult Create()
{
 _context.Add(new Person() { FirstName = "Nathan", LastName = "Owen" });
 _context.SaveChanges();
 return RedirectToAction(nameof(Index));
}
```

### Deleting Entities

To delete an entity from the database, you can use the Entity Framework context object. When you delete an entity from a database, the context marks the change tracking status of the entity as **Deleted**. When you call the **SaveChanges** method, the Entity Framework context object deletes the entity from the database.

The following code example shows how to delete an entity from a database by using the Entity Framework context object:

### Deleting an Entity

```
public IActionResult Delete(int id)
{
 var person = _context.People.SingleOrDefault(m => m.PersonId == id);
 _context.People.Remove(person);
 _context.SaveChanges();
 return RedirectToAction(nameof(Index));
}
```

### Updating Entities

To update an entity in the database, you can use the Entity Framework context object. When you update an entity, the context marks the change tracking status of the entity as **Modified**. When you call the **SaveChanges** method, the Entity Framework context object updates the entity in the database.

The following code example shows how to retrieve and update an entity by using the Entity Framework context object:

### Updating an Entity

```
public IActionResult Edit(int id)
{
 var person = _context.People.SingleOrDefault(m => m.PersonId == id);
 person.FirstName = "Brandon";
 _context.Update(person);
 _context.SaveChanges();
 return RedirectToAction(nameof(Index));
}
```

## Demonstration: How to Use Entity Framework Core

In this demonstration, you will learn how to use EF Core in an ASP.NET Core MVC application. You will see how to create an Entity Framework context, how to connect it to a SQLite database, how to use it in a controller, and how to use it to manipulate data.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use Entity Framework Core" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD07\\_DEMO.md#lesson-2-working-with-entity-framework-core](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD07_DEMO.md#lesson-2-working-with-entity-framework-core).

## Lesson 3

# Using Entity Framework Core to Connect to Microsoft SQL Server

You can use an Entity Framework context to connect to different databases. In the previous lesson, you saw how to use an Entity Framework context to connect to the SQLite database. In this lesson, you will see how to use an Entity Framework context to connect to a SQL Server database. The database to which the Entity Framework context is mapped is determined by a connection string. While you can store the connection string as a hard-coded string in your code, a better approach is to store the connection string in a configuration file.

When working with Entity Framework context and the entities it contains, you might need to update them from time to time. Changing the schema of the context or the entities might cause them not to be in sync with the tables and columns in the database. To ensure that the context and entities are in sync with the tables and columns in the database, you can use Migrations.

## Lesson Objectives

After completing this lesson, you will be able to:

- Connect an ASP.NET Core application to SQL Server by using EF Core.
- Use configuration in ASP.NET Core applications.
- Use a configuration file to store a connection string.
- Explain and use the Repository Pattern.
- Use migrations.

## Connecting to Microsoft SQL Server

You can use an Entity Framework context to connect to different databases. In the previous lesson, you saw how to use an Entity Framework context to connect to the SQLite database. In this topic, you will see how to use an Entity Framework context to connect to a SQL Server database.

The following code shows an entity class:

### An Entity Class

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
}
```

The **UseSqlServer** method configures the Entity Framework context to connect to a SQL Server database

```
public void ConfigureServices(IServiceCollection services)
{
 string connectionString = ...;
 services.AddDbContext<HrContext>(
 options => options.UseSqlServer(connectionString));
 services.AddMvc();
}
```

The following code example shows an Entity Framework context:

### The HrContext Class

```
public class HrContext : DbContext
{
 public HrContext(DbContextOptions<HrContext> options) : base(options)
 {
 }

 public DbSet<Person> Candidates { get; set; }

 protected override void OnModelCreating(ModelBuilder modelBuilder)
 {
 modelBuilder.Entity<Person>().HasData(
 new { PersonId = 1, FirstName = "James", LastName = "Smith" },
 new { PersonId = 2, FirstName = "Arthur", LastName = "Adams" }
);
 }
}
```

The following code example demonstrates how to configure HrContext to connect to a SQL Server database:

### Configuring HrContext to Connect to a SQL Server Database

```
public class Startup
{
 public void ConfigureServices(IServiceCollection services)
 {
 string connectionString =
"Server=(localdb)\MSSQLLocalDB;Database=HumanResources;Trusted_Connection=True;";
 services.AddDbContext<HrContext>(
 options => options.UseSqlServer(connectionString));
 services.AddMvc();
 }

 public void Configure(IApplicationBuilder app, HrContext ctx)
 {
 ctx.Database.EnsureDeleted();
 ctx.Database.EnsureCreated();
 app.UseMvc();
 }
}
```

The **UseSqlServer** method configures the Entity Framework context to connect to a SQL Server database. When using the connection string specified in the example above, the Entity Framework context connects to a database named **HumanResources** in the instance **LocalDB\mssqllocaldb**.

 **Note:** **LocalDB** is an extension of SQL Express that offers an easy way to create multiple database instances. Since it runs in user mode and starts on demand, it is easy to configure it.

When the **HumanResources** database is created, it contains a table named **Candidates**. The reason for this is that the **HrContext** class contains a property named **Candidates** of type **DbSet<Person>**. The **DbSet<>** properties of an Entity Framework context are mapped to tables in the database.

The **Candidates** table contains three columns:

- **PersonId** (PK, int, not null).
- **FirstName** (nvarchar(max), null).
- **LastName** (nvarchar(max), null).

The reason for this is that the **Person** entity contains the properties **PersonId**, **FirstName** and **LastName**. By default, each property of the **Person** entity is mapped to a column in the table. The name, data type and constraints of each column are determined by EF Core using a set of conventions.

 **Note:** The **PersonId** column is a primary key column in the database because the **PersonId** property is the entity key of the **Person** entity. By convention, the entity key of an entity in EF Core is either a property named **Id** or a property named **<type name>Id**.

## Configuration in EF Core

EF Core uses conventions to build a database based on the content of the Entity Framework context and the entities which belong to the Entity Framework context. In case you find that some of those conventions are not suitable for your needs, you can use configuration to override the conventions.

One way to specify configuration in EF Core is by using Fluent API. When you use Fluent API, you write the configuration in the **OnModelCreating** method of the Entity Framework context class.

For example, in EF Core you can use Fluent API to determine the name of a table that is created in a database. To determine the name of the table in the database you can use the **ToTable** method,

The following code example demonstrates how to create a table named **People** in the database which will be mapped to the **Candidates** property of the **HrContext** class:

### Naming Database Tables

```
public class HrContext : DbContext
{
 public HrContext(DbContextOptions<HrContext> options) : base(options)
 {}

 public DbSet<Person> Candidates { get; set; }

 protected override void OnModelCreating(ModelBuilder modelBuilder)
 {
 modelBuilder.Entity<Person>().ToTable("People");
 }
}
```

In addition to Fluent API, you can also specify the configuration in EF Core by using another method called data annotations. However, the data annotations method is quite different from the Fluent API method. While in the Fluent API method the configuration is specified in the **OnModelCreating** method of the Entity Framework context class, in the data annotations method the configuration is specified by using attributes in the entities.

The following code example demonstrates how to use data annotation to configure the **Person** entity:

### Using Data Annotations

```
public class Person
{
 [Key]
 public int MyId { get; set; }
```

```
[Required]
[StringLength(20)]
public string FirstName { get; set; }

[NotMapped]
public string LastName { get; set; }
}
```

When EF Core uses the **Person** entity above to create the table in the database, the table is created with two columns:

- **MyId** (PK, int, not null).
- **FirstName** (nvarchar(20), not null).

The **MyId** column in the database is a primary key column since the **MyId** property is the key of the **Person** entity. The **MyId** property is the key of the **Person** entity because it is annotated with the **Key** attribute.

The **FirstName** column in the database is not null since the **FirstName** property of the **Person** entity is annotated with the **Required** attribute. The data type of the **FirstName** column is **nvarchar(20)** since the maximum length of the **FirstName** property is configured to **20** by using the **StringLength** attribute.

The **LastName** property is annotated with the **NotMapped** attribute. Therefore, the **LastName** property is not mapped to a column in the database.

## Configuration in ASP.NET Core

Usually, an application needs to get parameters. These parameters are used to define settings and preferences which are needed when running the application. An example of such a parameter is the connection string which is used by the application to connect to a database. A recommended approach is to store such parameters in a configuration file. Storing the parameters in a configuration file will allow you to change the settings and preferences without recompiling the application.

- Configuration is stored in name-value pairs
- Configuration can be read from multiple sources
- To read data from a source, use a configuration provider
- Configuration providers exist for:
  - Files in JSON, XML and INI formats
  - Environment variables
  - Command line arguments
  - Custom provider
  - And more...

ASP.NET Core is capable of reading parameters from various sources. To access a source ASP.NET Core uses a configuration provider. There are configuration providers for several configuration file formats, including JSON, XML, and INI. In addition, there are configuration providers for other sources such as command line arguments and environment variables. You can even create a custom provider if needed.

In this topic, you will see how an ASP.NET Core application can read parameters from JSON files and XML files. The parameters in the files will be stored as name-value pairs.

### JSON Configuration

JSON is a text format which can be used for several purposes, such as transmitting data between applications and storing data in a file. One of the biggest advantages of JSON is that it is very easy for people to read and write JSON. JSON can also be parsed and generated easily by applications.

The following code example shows the content of a JSON file named config.json:

### A JSON Configuration File

```
{
 "firstName": "Mark",
 "lastName": "Smith",
 "address": {
 "streetAddress": "35 3rd Street",
 "city": "Miami"
 }
}
```

To use a JSON configuration file in ASP.NET Core, you should add the JSON configuration provider. To add the JSON configuration provider, you can call the **AddJsonFile** method on an object that implements the **IConfigurationBuilder** interface. The **AddJsonFile** method gets the path to the configuration file as a parameter. Then you can call the **Build** method on the object, which implements the **IConfigurationBuilder** interface, to build an object that implements the **IConfiguration** instance with settings from the set of sources registered in the **IConfigurationBuilder.Sources** property. Finally, you can call the **UseConfiguration** method, passing it the object which implements the **IConfiguration** instance as a parameter to use the given configuration settings on the web host.

The following code example demonstrates how to use the JSON configuration provider to load the values from the config.json file:

### Using JSON Configuration Provider in Program.cs

```
public class Program
{
 public static void Main(string[] args)
 {
 CreateWebHostBuilder(args).Build().Run();
 }

 public static IWebHostBuilder CreateWebHostBuilder(string[] args)
 {
 var builder = new ConfigurationBuilder()
 .SetBasePath(Directory.GetCurrentDirectory())
 .AddJsonFile("config.json");

 IConfiguration configuration = builder.Build();

 return WebHost.CreateDefaultBuilder(args)
 .UseConfiguration(configuration)
 .UseStartup<Startup>();
 }
}
```

 **Note:** The call to the **WebHost.CreateDefaultBuilder** method initializes a new instance of the **WebHostBuilder** class with pre-configured defaults. One of the defaults is loading values from a configuration file named **appsettings.json**. Therefore, if you need to load values from a file named **appsettings.json**, calling the **CreateDefaultBuilder** method is sufficient and you don't need to call the **AddJsonFile** method and pass to it **appsettings.json** as a parameter.

To load a value from the configuration file, you can use an indexer of an object which implements the **IConfiguration** interface and pass to it the name of the name-value pair as a parameter.

The following code example demonstrates how to retrieve a value from the config.json file. The value of the **value** variable will be **Mark**:

### Retrieve a Value from a Configuration File

```
public class SomeController : Controller
{
 private IConfiguration _configuration;

 public SomeController(IConfiguration configuration)
 {
 _configuration = configuration;
 }

 public IActionResult Index()
 {
 string value = _configuration["firstName"];
 return Content(value);
 }
}
```

If the name of the name-value pair is hierarchical, you should use a colon to separate the sections of the name to retrieve the value.

The following code example demonstrates how to retrieve a value from a hierarchical name. The value of the **value** variable will be **Miami**:

### Retrieve a Value from a Hierarchical Name

```
public class SomeController : Controller
{
 private IConfiguration _configuration;

 public SomeController(IConfiguration configuration)
 {
 _configuration = configuration;
 }

 public IActionResult Index()
 {
 string value = _configuration["address:city"];
 return Content(value);
 }
}
```

## XML Configuration

In addition to using JSON configuration files in ASP.NET Core, it is also possible to use XML configuration files in ASP.NET Core. To use an XML configuration file in ASP.NET Core, you should add the XML configuration provider. To add the XML configuration provider, you can call the **AddXmlFile** method on an object which implements the **IConfigurationBuilder** interface.

You can configure an ASP.NET Core application to read parameters from multiple configuration files. To do that you should add several configuration providers. For example, in case you want to read parameters from both JSON and XML files, you should call both **AddJsonFile** and **AddXmlFile** on an object which implements the **IConfigurationBuilder** interface. Notice that in case you read data from multiple sources, the order in which the data providers are added matter. In case the same name exists in multiple sources, only the value from the last source will be returned. Values from other sources will be ignored.

The following code example shows the content of an XML file named config.xml:

### An XML Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<person>
 <firstName>James</firstName>
 <age>23</age>
</person>
```

The following code example demonstrates how to use the XML configuration provider to load the values from the config.xml file. Notice that the XML configuration provider is added after the JSON configuration provider:

### Using an XML Configuration Provider in Program.cs

```
public class Program
{
 public static void Main(string[] args)
 {
 CreateWebHostBuilder(args).Build().Run();
 }

 public static IWebHostBuilder CreateWebHostBuilder(string[] args)
 {
 var builder = new ConfigurationBuilder()
 .SetBasePath(Directory.GetCurrentDirectory())
 .AddJsonFile("config.json")
 .AddXmlFile("config.xml");

 IConfiguration configuration = builder.Build();

 return WebHost.CreateDefaultBuilder(args)
 .UseConfiguration(configuration)
 .UseStartup<Startup>();
 }
}
```

The following code example demonstrates how to retrieve values from the config.json and config.xml files:

### Retrieve Values from Configuration Files

```
public class SomeController : Controller
{
 private IConfiguration _configuration;

 public SomeController(IConfiguration configuration)
 {
 _configuration = configuration;
 }

 public IActionResult Index()
 {
 var firstName = _configuration["firstName"];
 var lastName = _configuration["lastName"];
 var age = _configuration["age"];
 return Content($"{firstName} {lastName} is {age} years old");
 }
}
```

In the code example above, the value of the **age** variable will be **23**. This value is read from the **config.xml** file. The value of the **lastName** variable will be **Smith**. This value is read from the **config.json** file. The value of the **firstName** variable will be **James**. This value is read from the **config.xml** file.

 **Note:** The **firstName** name appears in both the **config.json** and **config.xml** files. The reason that the value returned is the value from the **config.xml** file is because the XML configuration provider is added after the JSON configuration provider.

## Specifying a Connection String in a Configuration File

Storing the connection string for your application in a configuration file instead of storing it as a hard-coded string in your code is a good practice. In older versions of ASP.NET, all configurations including connection strings were stored in the **web.config** file. In ASP.NET Core, you can store the connection strings in other configuration files in various formats, including JSON, XML and INI. You can also store the connection strings in other sources such as environment variables or command line arguments.

- Connection string in a configuration file:

```
{
 "ConnectionStrings": {
 "DefaultConnection": "..."
 }
}
```

- Reading the connection string from the configuration file:

```
string connectionString =
 _configuration.GetConnectionString("DefaultConnection");
services.AddDbContext<HrContext>(options =>
 options.UseSqlServer(connectionString));
```

The following code shows how to store a connection string in a JSON configuration file:

### Storing a Connection String in a JSON Configuration File

```
{
 "ConnectionStrings": {
 "DefaultConnection":
 "Server=(localdb)\MSSQLLocalDB;Database=HumanResources;Trusted_Connection=True"
 }
}
```

The following code example demonstrates how to read the connection string from the configuration file by using an indexer of **IConfiguration**. The connection string is then used to connect the Entity Framework context to a database named **HumanResources** in the **LocalDB\MySQLLocalDB** instance:

### Reading a Connection String from a Configuration File

```
public class Startup
{
 private IConfiguration _configuration;

 public Startup(IConfiguration configuration)
 {
 _configuration = configuration;
 }

 public void ConfigureServices(IServiceCollection services)
 {
 string connectionString = _configuration["ConnectionStrings:DefaultConnection"];
 services.AddDbContext<HrContext>(options =>
 options.UseSqlServer(connectionString));
 services.AddMvc();
 }

 public void Configure(IApplicationBuilder app, HrContext ctx)
 {
 ctx.Database.EnsureDeleted();
 ctx.Database.EnsureCreated();
 app.UseMvc();
 }
}
```

```
}
```

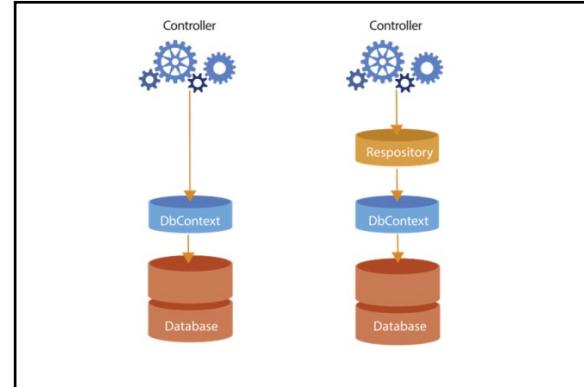
To read a connection string from a configuration file, you can replace the call to the indexer of **IConfiguration** with a call to the **GetConnectionString** method. The following code example demonstrates how to use the **GetConnectionString** method to read the connection string from the configuration file:

### Using the GetConnectionString Method

```
public void ConfigureServices(IServiceCollection services)
{
 string connectionString = _configuration.GetConnectionString("DefaultConnection");
 services.AddDbContext<HrContext>(options => options.UseSqlServer(connectionString));
 services.AddMvc();
}
```

## The Repository Pattern

Web applications usually need to access databases and typically, the simplest approach is to use the controller to access the database directly. However, calling the database directly from the controller is not always the best practice. A more flexible approach is that the controller will call a repository that will access the database. Using this approach, the controller will not access the database directly. To call a repository from a controller, you should perform the following steps:



1. Define an interface for the repository class. This interface declares the methods that the repository class uses to read and write data from and to the database.
2. Create and write code for the repository class. This class must implement all the data access methods declared in the interface.
3. Use dependency injection to inject the repository class to a controller.
4. Modify the controller class to use the repository class.

The following code shows a repository interface:

### A Repository Interface

```
public interface IRepository
{
 IEnumerable<Person> GetPeople();
 void InsertPerson();
 void DeletePerson();
 void UpdatePerson();
}
```

The following code shows a repository class:

### A Repository Class

```
public class MyRepository : IRepository
{
 private HrContext _context;

 public MyRepository(HrContext context)
 {
 _context = context;
 }

 public IEnumerable<Person> GetPeople()
 {
 return _context.Candidates;
 }

 public void InsertPerson()
 {
 _context.Candidates.Add(new Person() { FirstName = "John", LastName = "Smith" });
 _context.SaveChanges();
 }

 public void DeletePerson()
 {
 var person = _context.Candidates.FirstOrDefault(c => c.LastName == "Smith");
 _context.Candidates.Remove(person);
 _context.SaveChanges();
 }

 public void UpdatePerson()
 {
 var person = (from c in _context.Candidates where c.FirstName == "James" select
c).Single();
 person.FirstName = "Mike";
 _context.SaveChanges();
 }
}
```

The following code example demonstrates how to register the repository in the **ConfigureServices** method in the Startup.cs file:

### Registering the Repository

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddTransient< IRepository, MyRepository>();
 string connectionString = _configuration.GetConnectionString("DefaultConnection");
 services.AddDbContext< HrContext >(options => options.UseSqlServer(connectionString));
 services.AddMvc();
}
```

The following code example demonstrates how a controller can use the repository:

### A Controller Using the Repository

```
public class HrController : Controller
{
 IRepository _repository;

 public HrController(IRepository repository)
 {
 _repository = repository;
 }
```

```
[Route("Hr/Index")]
public IActionResult Index()
{
 var list = _repository.GetPeople();
 return View(list);
}

[Route("Hr/Insert")]
public IActionResult Insert()
{
 _repository.InsertPerson();
 return RedirectToAction("Index");
}

[Route("Hr/Delete")]
public IActionResult Delete()
{
 _repository.DeletePerson();
 return RedirectToAction("Index");
}

[Route("Hr/Update")]
public IActionResult Update()
{
 _repository.UpdatePerson();
 return RedirectToAction("Index");
}
}
```

## Demonstration: How to Apply the Repository Pattern

In this demonstration, you will learn how to use a repository in an MVC application to connect to a SQL Server database. You will see how to create a configuration file and store a connection string in the configuration file. You will also see how to inject a repository to a controller and use the repository in the controller to access the database.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Apply the Repository Pattern" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD07\\_DEMO.md#lesson-3-using-entity-framework-core-to-connect-to-microsoft-sql-server](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD07_DEMO.md#lesson-3-using-entity-framework-core-to-connect-to-microsoft-sql-server).

## Using Migrations

Working with an application that interacts with a database requires you to create models and sometimes, you might find yourself changing or upgrading those models according to the application needs. When working with EF Core, your database is relying on the models you have created. To reflect the changes in the models to the database, you can use migrations. Using migrations, you will be able to create a database, upgrade it and manipulate it according to your application models.

- Migrations enable applying schema changes to the database
- You can work with migrations by using the Entity Framework Core Package Manager Console (PMC) Tools
- Fundamental migration commands
- **Add a migration:**  
Add-Migration <name\_of\_the\_migration>
- **Apply the migration to the database:**  
Update-Database

If you created your database by using an Entity Framework context, and you later want to change something in your domain model classes e.g. your schemas, EF Core will not update the database automatically. You might encounter exceptions while running queries or saving your changes to the database. You can use migrations to update the database schema automatically to match the changes you made in your classes without having to recreate the database.

With migrations, you define the initial state of your classes and your database. After you change your classes and execute the migrations in design time, the set of changes you performed over your classes is translated to the required migration steps for the database, and then those steps are generated as database instructions in the code. You can apply the changes to the database in design-time, before deploying the version of the application. Alternatively, you can have the application execute the migration code after it starts.

## Entity Framework Core Package Manager Console Tools

You can work with migrations by using the Entity Framework Core Package Manager Console (PMC) Tools. You can use NuGet's Package Manager Console to run the PMC tools in Visual Studio.

To create a database by using migrations, you should first add an initial migration. Then, you need to apply the migration to the database. As a result, the schema of the database will be created.

The following code example demonstrates how to add an initial migration:

### Adding an Initial Migration

```
Add-Migration InitDatabase
```

The following code example shows which command you should run to apply the migration to create the schema of the database:

### Create the Schema of the Database

```
Update-Database
```

When you change an entity, the database and the code are not in sync anymore. To sync the code and the database, you should add another migration.

The following code example demonstrates a change in the **Person** entity. A new property named **Age** is added to the **Person** entity. Therefore, the database and code are not synced anymore:

### Updating the Person Entity

```
public class Person
{
 public int PersonId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public int? Age { get; set; }
}
```

The following code example demonstrates how to add a new migration:

### Adding a New Migration

```
Add-Migration AddAge
```

To apply the **AddAge** migration to the database, use the following command:

#### Applying the AddAge Migration to the Database

```
Update-Database
```

After applying the **AddAge** migration, a new column named **Age** is added to the database.

# Lab: Using Entity Framework Core in ASP.NET Core

## Scenario

Your company is planning to write a web application for managing cupcakes and bakeries. To connect the application to a database, your development team has decided to use Entity Framework Core. You have been asked to create a class that derives from a **DbContext** class, and then use the class to retrieve data from the database and store data in the database. The application will enable users to store uploaded cupcakes, edit their properties, view their details, and delete them in

## Objectives

After completing this lab, you will be able to:

- Add new models to the application and add properties to the models.
- Add a class that derives from the **DbContext** class.
- Use a repository to the application to connect to the database.
- Use Entity Framework Core to retrieve and store data.
- Use Migrations to create a database and to update the schema of the database.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD07\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD07_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD07\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD07_LAK.md).

## Exercise 1: Adding Entity Framework Core

### Scenario

In this exercise, you will first add the **Cupcake** model and the **Bakery** model to the **Cupcake** web application. You will then add an Entity Framework context class named **CupcakeContext** to the web application. After that, you will configure the **CupcakeContext** class to connect to a SQLite database. Finally, you will use data seeding to populate the database with sample data when the database is created.

The main tasks for this exercise are as follows:

1. Create model classes.
2. Create a class that derives from DbContext.
3. Set up Entity Framework Core to use SQLite.
4. Use OnModelCreating to populate the database.

## Exercise 2: Use Entity Framework Core to Retrieve and Store Data

### Scenario

In this exercise, you will first create a repository for the web application. The repository will access a SQLite database by using Entity Framework Core. You will then use dependency injection to inject the service to a controller. You will use the repository in the controller to access the database. In the controller, you will retrieve cupcakes and bakeries data, and then you will manipulate the data.

The main tasks for this exercise are as follows:

1. Create a repository.
2. Update a controller to use a repository.
3. Use Entity Framework Core to retrieve data.
4. Manipulate data by using Entity Framework Core.
5. Run the application.

## Exercise 3: Use Entity Framework Core to Connect to Microsoft SQL Server

### Scenario

In this exercise, you will first configure the Cupcakes Shop web application to connect to an SQL Server database instead of connecting to an SQLite database. You will then store the connection string which is used to connect to the database in a configuration file. After that, you will use Migrations to create the database. Finally, you will add a property to an entity, and use Migrations to update the database schema.

The main tasks for this exercise are as follows:

1. Connect to a Microsoft SQL Server.
2. Specify a connection string in a configuration file.
3. Use Migrations to create a database.
4. Run the application.
5. Use Migrations to update the database schema.
6. Run the application.

**Question:** A developer in your team added a property to the **Bakery** class. When he ran the application, an exception was raised. Can you suggest to him how to solve the problem?

**Question:** How can you change the application to display the cupcakes ordered by their Price?

# Module Review and Takeaways

In this module, you have learned about the need for a database in a web application and how an application can communicate with a database. You learned how to bind model classes to database tables by using EF Core and how to query data in the database by writing LINQ. You also learned how to manipulate the data in the database by using EF Core.

To connect an Entity Framework context to a database you should use a database provider. In this module, you saw how to connect a web application to two different kinds of databases – SQLite and SQL Server. You also learned the importance of storing a connection string in a configuration file.

During development properties of entities might be changed. As a result, the database might not be in sync with the code anymore. To solve this, you can use migrations. In this module, you learned what are migrations and how you can use migrations to save the code and the database in sync.

## Review Questions

**Question:** Why should you use EF Core instead of directly manipulating the database by using SQL statements in ADO.NET?

**Question:** You want to connect your application to a SQL Server database. How can you do it?

## Best Practice

It is recommended to store connection strings in a configuration file instead of storing them as hard-coded strings.

## Common Issues and Troubleshooting Tips

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



# Module 8

## Using Layouts, CSS and JavaScript in ASP.NET Core MVC

### Contents:

Module Overview	08-1
<b>Lesson 1: Using Layouts</b>	<b>08-2</b>
<b>Lesson 2: Using CSS and JavaScript</b>	<b>08-8</b>
<b>Lesson 3: Using jQuery</b>	<b>08-19</b>
<b>Lab: Using Layouts, CSS and JavaScript in ASP.NET Core MVC</b>	<b>08-29</b>
Module Review and Takeaways	08-31

## Module Overview

While building web applications, you should apply a consistent look and feel to the application. You should include consistent header and footer sections in all the views. Microsoft ASP.NET Core MVC includes features such as cascading style sheets (CSS) styles and layouts that enhance the appearance and usability of your web application.

In ASP.NET Core MVC, you can create interactive HTML elements by using JavaScript. You need to know how to use JavaScript in your web application. To simplify adding JavaScript to your web application, you need to know how to use libraries such as jQuery.

### Objectives

After completing this module, you will be able to:

- Apply a consistent layout to ASP.NET Core MVC applications.
- Add JavaScript code to your web application.
- Use the jQuery library in your web application.

# Lesson 1

## Using Layouts

You need to build multiple views to support the features of your application, such as creating an order and querying order history. Some of these views can have similar parts. You can put these parts into the content of the views. However, this can result in maintenance issues. This is because, whenever you have to update the parts, you have to update each view.

To resolve these maintenance issues, you can build a common module or a shared view. A shared view that helps to store the application logic is called a *layout*. ASP.NET Core MVC includes features that help simplify the process of creating and using layouts. You can further simplify the application management process by using the \_ViewStart file that helps you to apply a layout to each view, instead of individually editing each view.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe layouts.
- Describe how to use layouts.
- Describe the \_ViewStart file.
- Describe how to use sections in a layout.

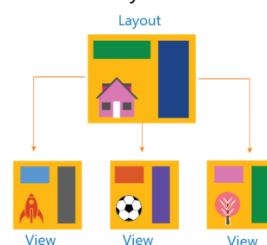
### What are Layouts?

The ASP.NET Core MVC Razor engine includes a feature called layouts. Layouts are also called template views. Layouts enable you to define a common style template and then apply it to all the views in a web application. The functionality of layouts is similar to that of the master page in a traditional ASP.NET web application. You can use layouts to define the content layout or logic that is shared across views.

You can define multiple layouts in an ASP.NET Core MVC application, and each layout can have multiple sections. You can define these sections anywhere in the layout file, even in the <head> section of the HTML. Sections enable you to output dynamic content to multiple, non-contiguous regions of the final response that is sent to the browser.

You can use layouts to:

- Create a style template for a web application
- Define the content layout to share across all views



## Creating a Layout

When you create layouts, you should store the layout files in the **\Views\Shared** folder of the project. The **\Views\Shared** folder is the default location where you can store common view files or templates.

The following example illustrates a layout named **\_Layout.html**:

### A Layout view

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
</head>
<body>
 <div>
 @RenderBody()
 </div>
</body>
</html>
```

When you create a layout:

- You should store the layout files in the **\Views\Shared** folder
- You can use the **@RenderBody** method to place the content of a view in the layout
- You can use the **ViewBag** property to pass information between a view and the layout

In the preceding example, the **@RenderBody** method indicates to the rendering engine where the content of the view should be placed.

The layout and view share the same **ViewBag**. You can use the **ViewBag** property to pass information between a view and a layout. To pass information, you can add a property to the **ViewBag** property in the View file and use the same property in the layout file. Properties help you control the content in the layout to dynamically render webpages from the code in the view file. For example, consider that a template uses the **ViewBag.Title** property to render the **<title>** content in the view. This property helps you to not only define the **Title** property of the **ViewBag** property in the view but also to retrieve the property in the layout. This retrieval is possible because the code in the view file runs before the layout.

## Linking Views and Layouts

After defining a layout, you can link the layout to the view files. To link a layout to a view, you should add the **Layout** directive at the top of the view file.

The following code shows a simple controller, which calls a view that has a link to a layout:

### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 return View();
 }
}
```

To link views and layouts:

- You can add the **Layout** directive at the top of the view file
- You can use the **\_ViewStart.cshtml** file to define the layout
  - Add the **\_ViewStart.cshtml** file to the **\Views** folder of your project

The following code shows a view named **Index** that has a link to the **\_Layout** layout:

### A view with a link to a layout

```
@{
 ViewBag.Title = "Home Page";
 Layout = "_Layout";
}

<h2>Index</h2>
```

If a user requests the relative URL **/Home/Index**, the following response will be sent to the browser:

### Text sent to the browser

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Home Page</title>
</head>
<body>
 <div>
 <h2>Index</h2>
 </div>
</body>
</html>
```

You can use the **ViewBag.Title** property to pass the page title information from the view to the layout.

You can define other properties along with the **ViewBag** property, such as **<meta>** elements in the **<head>** section and enable them to pass information to the layout.

### The **\_ViewStart.cshtml** File

Usually, you have the same layout across an entire web application. You can define the layout for an application by using the **\_ViewStart.cshtml** file. During run time, the code in the **\_ViewStart.cshtml** file runs before all the other views in the web application. Therefore, you can place all common application logic in the **\_ViewStart.cshtml** file.

To use the **\_ViewStart.cshtml** file, you should add it to the **\Views** folder of your project. The following code illustrates the content of the **\_ViewStart.cshtml** file:

### The **\_ViewStart.cshtml** file

```
@{
 Layout = "_Layout";
}
```

If the web application contains the **\_ViewStart.cshtml** file shown above, you don't have to include the link to the layout in the **Index** view.

The following code shows the **Index** view, without the link to the layout:

### The **Index** view

```
@{
 ViewBag.Title = "Home Page";
}

<h2>Index</h2>
```

If a user requests the relative URL **/Home/Index**, the following response will be sent to the browser:

#### Text sent to the browser

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Home Page</title>
</head>
<body>
 <div>
 <h2>Index</h2>
 </div>
</body>
</html>
```

## Using Sections in a Layout

A section is a region of content within a layout. It is a placeholder within a layout that allows a view to insert content. You can add sections to a layout by using the **RenderSection** method. The **RenderSection** method gets the name of the section as a parameter. In a view, you can set the content of the section by using the **@section** directive followed by the name of the section.

The following example illustrates a layout that contains sections:

- Use the **@RenderSection** method in a layout

```
@RenderSection("section1")
@RenderBody()
@RenderSection("section2", false)
```

- Use the **@section** directive in a view

```
@section section1
{
 <div>This is section1 content</div>
}
<div>This is the body</div>
```

#### Using sections in a layout

```
<!DOCTYPE html>
<html>
<head>
 <title></title>
</head>
<body>
 @RenderSection("section1")
 <div>This is layout content between section1 and the body</div>
 @RenderBody()
 <div>This is layout content between the body and section2</div>
 @RenderSection("section2")
</body>
</html>
```

The **section1** parameter in the **RenderSection** method specifies the name of the section.

The following code shows a simple controller, which calls a view that has a link to a layout:

#### A Controller class

```
public class HomeController : Controller
{
 [Route("Home/Index")]
 public IActionResult Index()
 {
 return View();
 }
}
```

The following code shows a view named **Index** that has a link to the `_Layout` layout. It demonstrates how to use the `@section` directive:

### Using the `@section` directive

```
@{
 Layout = "_Layout";
}

@section section1
{
 <div>This is section1 content</div>
}

<div>This is the body</div>

@section section2
{
 <div>This is section2 content</div>
}
```

If a user requests the relative URL `/Home/Index`, the following text will be displayed in the browser:

### Text displayed in the browser

```
This is section1 content
This is layout content between section1 and the body
This is the body
This is layout content between the body and section2
This is section2 content
```

## The Required Parameter

The `RenderSection` method gets the name of the section as a parameter. It can get another parameter that indicates if the section is required. This is an optional parameter of type `bool`. Consider a section in a layout that is required and you do not have a corresponding `@section` directive in the view file. In this case, an exception will be thrown at run time.

The following code shows a view named **Index** in which the `@section section2` block is commented:

### Not implementing a section

```
@{
 Layout = "_Layout";
}

@section section1
{
 <div>This is section1 content</div>
}

<div>This is the body</div>

@* @section section2
{
 <div>This is section2 content</div>
}*@
```

If a user requests the relative URL `/Home/Index`, an exception will occur. The reason is that although the `section2` section is declared as required in the layout, there is no corresponding section in the view.

The following code shows how to set the **section2** section as optional in a layout:

### Setting an optional section

```
<!DOCTYPE html>
<html>
<head>
 <title></title>
</head>
<body>
 @RenderSection("section1")
 <div>This is layout content between section1 and the body</div>
 @RenderBody()
 <div>This is layout content between the body and section2</div>
 @RenderSection("section2", false)
</body>
</html>
```

If a user requests the relative URL **/Home/Index**, the following text will display in the browser:

### Text displayed in the browser

```
This is section1 content
This is layout content between section1 and the body
This is the body
This is layout content between the body and section2
```

## Demonstration: How to Create a Layout and Link it to a View

In this demonstration, you will learn how to add a layout and how to use the **@RenderBody** and **@RenderSection** methods in the layout. You will see how to add a **\_ViewStart.cshtml** file to an MVC application. You will also see how to connect views to a layout by using the **\_ViewStart.cshtml** file.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Create a Layout and Link it to a View" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD08\\_DEMO.md#demonstration-how-to-create-a-layout-and-link-it-to-a-view](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_DEMO.md#demonstration-how-to-create-a-layout-and-link-it-to-a-view).

## Lesson 2

# Using CSS and JavaScript

While HTML defines the structure and context of a web page, CSS defines how it should appear in a browser. You need to know how to import styles into a web application to ensure consistency in the appearance of the application. It is also important to know how to add a link to an external CSS file from a layout.

You can create interactive HTML elements in your web application by using JavaScript. In ASP.NET Core you can add these interactive elements to views and layouts. It is important to know how to add JavaScript code to your web application.

Developing the CSS and JavaScript of your web application from scratch is not always the best option. There are many external libraries that you can use in a web application to improve its quality and to shorten the development time. In this lesson, you will see several options to add external libraries to an ASP.NET Core application.

### Lesson Objectives

After completing this lesson, you will be able to:

- Import styles into an ASP.NET Core MVC web application.
- Apply a consistent look and feel to an MVC web application.
- Describe how to add JavaScript files to an MVC web application.
- Describe how to use external libraries in ASP.NET Core MVC web applications.

### Importing Styles

Cascading Style Sheets (CSS) is an industry standard for applying styles to HTML pages. After creating CSS styles, you should import these styles into the web application. Different methods of applying CSS to a webpage are available. These methods include external CSS file, inline CSS, and CSS code block in HTML. Developers usually use an external CSS file because this file is shared across multiple pages and it helps to apply a consistent style across the application. It is common to add a link to the external CSS file from a layout. Therefore, after importing the CSS file into the web application, you need to modify the layout of the web application, so that you can use the CSS styles that you imported. You can modify the layout of a web application by using the `<link>` element.

After importing the CSS file:

- You should modify the layout of the web application by using the `<link>` element
- You can add CSS selectors to define how the styles should be applied:
  - CSS class selector helps specify a style for a group of elements
  - CSS id selector helps specify a style for any unique element in the HTML code

```
.menu
{
 font-weight:bold;
}
→ <p class="menu"> this is menu</p>
```

The following example shows how to use the `<link>` element:

#### Linking a layout to a CSS file

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <link href="~/style.css" rel="stylesheet" />
```

```
</head>
<body>
 <div>
 @RenderSection("MenuBar")
 </div>
 <div>
 @RenderBody()
 </div>
</body>
</html>
```

 **Note:** The **style.css** file should be located in the **wwwroot** folder. You should also add the **UseStaticFiles** middleware to the **Configure** method of the **Startup** class. Working with static files is covered in Module 3, "Configuring Middleware and Services in ASP.NET Core".

CSS selectors help browsers to determine how the CSS styles should be applied. You can use various selectors, such as class and id selectors to apply styles to HTML elements.

## CSS class Selector

You can define a CSS class selector to specify a style for a group of elements. To apply the class selector to an HTML element, you need to add the **class** attribute to the HTML element. You can use the **.<class>** syntax to add the style in the CSS file.

The following example shows how to add the **class** attribute in a view:

### Using a class attribute in a view

```
@{
 ViewBag.Title = "Details";
 Layout = "_Layout";
}

<h2>Details</h2>

@section MenuBar {
 <p class="menu"> this is menu</p>
}
```

The following CSS snippet shows how to create the class selector:

### Applying a style to a class

```
.menu {
 font-weight:bold;
}
```

## CSS id Selector

You can use the CSS id selector to specify a style for any unique element in your HTML code. To apply the id selector to an HTML element, you need to add the **id** attribute and a unique name to the HTML element. You can use the **#<id>** syntax to add the style in the CSS file.

The following example shows how to use the **id** attribute in a view:

### Using an id attribute in a view

```
@{
 ViewBag.Title = "Details";
 Layout = "_Layout";
}

<h2>Details</h2>

@section MenuBar {
 <p id="leftmenu"> this is menu</p>
}
```

The following CSS snippet shows how to create the CSS id selector:

### Creating an id selector

```
#leftmenu {
 font-size:16px;
}
```

## Rendering and Executing JavaScript Code

JavaScript code helps add interactive functionalities to the webpages of your application. You can add JavaScript code to web applications by:

- Adding the JavaScript code to a view file.
- Defining the JavaScript code in dedicated JavaScript files.

The following example shows how to add JavaScript code to a view file:

- You can add JavaScript code to add interactive functionalities to webpages

```
<script>
 function helloWorld() {
 alert('Hello World');
 }
</script>
```

- You can add JavaScript code to web applications by:

- Adding the JavaScript code to a view
- Defining the JavaScript code in dedicated JavaScript files

### Inserting a JavaScript function

```
@{
 Layout = null;
}

<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Index</title>
 <script>
 function helloWorld() {
 alert('Hello World');
 }
 </script>
</head>
<body>
 ...
</body>
</html>
```

If you have multiple views in a web application, you need to add the JavaScript code for each view. You cannot simultaneously add JavaScript code for multiple views. Therefore, you can define the JavaScript code in a JavaScript file (.js file). Then, you can reference the JavaScript file in multiple views. This enables you to maintain a single JavaScript file to edit the JavaScript code for multiple views. You can also have a reference to multiple JavaScript code files from a single view.

The following example shows the content of a JavaScript file that contains a function:

### The example.js file

```
function helloWorld() {
 alert('Hello World');
}
```

 **Note:** The **example.js** file should be located in the **wwwroot** folder. You should also add the **UseStaticFiles** middleware to the **Configure** method of the **Startup** class. Working with static files is covered in Module 3, "Configuring Middleware and Services in ASP.NET Core".

The following example shows how to reference a JavaScript file in a view:

### Referencing a JavaScript file

```
@{
 Layout = null;
}

<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Index</title>
 <script src="~/example.js"></script>
</head>
<body>
 ...
</body>
</html>
```

 **Note:** You can create a **Scripts** folder in the **wwwroot** folder of your MVC project, and then save all JavaScript files in the **Scripts** folder.

### Referencing a JavaScript File from a Layout

In ASP.NET Core MVC applications, it is common to add a reference to the JavaScript file from a layout. Therefore, after importing the JavaScript file into the web application, you need to modify the layout of the web application, so that you can use the JavaScript code. You can modify the layout of a web application, by using the **<script>** element.

The following example shows how to reference a JavaScript file in a layout:

### Referencing JavaScript file in a layout

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <script src="~/example.js"></script>
</head>
```

```
<body>
 @RenderBody()
</body>
</html>
```

## Calling JavaScript Functions

You can call functions defined in JavaScript files by using script blocks or event handlers.

The following code example demonstrates how to call the **helloWorld** function from a view:

### Using a script block to call a function

```
@{
 Layout = "_Layout";
}
<h2>Index</h2>
<script>
 helloWorld();
</script>
```

If you want to avoid calling the JavaScript function directly, you can use the **onclick** JavaScript event to trigger JavaScript functions. The **onclick** event initiates the JavaScript function when you click the corresponding HTML element. JavaScript functions that are attached to the document object model (DOM) events are called event handlers.

The following code shows how to add the **helloWorld** event handler to the button's **onclick** event:

### Using an event handler in a view

```
@{
 Layout = "_Layout";
}

<h2>Index</h2>
<input type="button" value="Hello" onclick="helloWorld();"/>
```

## Calling JavaScript Built-In Functions

JavaScript has a large variety of built-in functions that allow you to perform different actions. There are functions that deal with converting one type of variable to another, functions that perform mathematic calculations and there are many others.

The following functions are examples of functions that deal with converting one type of variable to another:

- **parseInt()**. This function accepts a string and returns an integer. For example, the string "20.6" will be converted to 20.
- **parseFloat()**. This function accepts a string and returns a floating point number. For example, the string "50.1" will be converted to 50.1.

In addition to built-in functions, there are also built-in objects in JavaScript. For example, there is a built-in object in JavaScript named **Math**. **Math** is an object that allows to perform mathematic calculations on numbers. It includes many built in functions, which include:

- **Math.round()**. A function that is used to round off a number to the nearest integer.
- **Math.floor()**. A function that is used to round down a number to the nearest integer.
- **Math.ceil()**. A function that is used to round up a number to the nearest integer.

The following code example demonstrates how to use some of the built-in JavaScript functions to perform mathematic calculations:

### Using built-in JavaScript functions

```
var userInput = "50.5";
var parsedNumber = parseFloat(userInput); // Output 50.5
var roundedNumber = Math.round(parsedNumber); // Output 51
var flooredNumber = Math.floor(parsedNumber); // Output 50
var ceiledNumber = Math.ceil(parsedNumber); // Output 51
```

## Using External Libraries

If you want to add client-side logic to your application and create interactive elements such as sliders, tabs, and popups, you will need code that allows you to implement them. Through the years, developers have created a vast number of libraries that solve common development problems and allow developers to shorten their code, speed-up the development cycle and avoid duplication.

There are many common libraries that are frequently used such as jQuery. jQuery helps you to perform tasks such as accessing and modifying HTML elements on a webpage and creating commonly used interactive elements.

To add a library to your application, you can:

- Download the source files from an official source
- Use a CDN (Content Delivery Network)
- Use a Package Manager
  - NuGet – For server-side libraries
  - Yarn
  - Webpack
  - Bower
  - npm

 **Note:** jQuery and its syntax will be covered in greater detail in Lesson 3, "Using jQuery".

To utilize jQuery and other libraries in an ASP.NET Core application, you first need to import them into the project. There are many ways to import a library into the application, but some are more recommended than others.

### Serving Files from a Local Folder

One of the ways to add a library to an ASP.NET Core application is to download the source files from an official source such as jQuery from the jQuery website or Bootstrap from the Bootstrap website.

 **Note:** Bootstrap is covered in Module 9, "Client-Side Development".

After downloading the files, in order to load them in the application, you should add them to the static files folder which by default is the **wwwroot** folder.

 **Best Practice:** You should create a scripts folder in the **wwwroot** folder of your ASP.NET Core project and then store all the JavaScript libraries there. Similarly, you should create a styles folder in the **wwwroot** folder of your ASP.NET Core project and then store all the CSS libraries there.

The following code example demonstrates how to link a layout to the jQuery library.

### Linking a layout to the jQuery library

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <link href("~/css/site.css" rel="stylesheet" />
 <script src "~/js/jquery-3.3.1.js"></script>
</head>
<body>
 <div>
 @RenderBody()
 </div>
</body>
</html>
```

One of the disadvantages of serving files from a local folder is that the loading time of those files is long. To make the loading time shorter you can serve the files by using a Content Delivery Networks (CDN) instead of serving them from a local folder.

### Serving Files using CDN

A CDN is a group of geographically distributed servers used for hosting content for web applications. In many cases, you can bring web content geographically closer to your application's users by using a CDN to host libraries. This will also improve the scalability and robustness of the delivery of that content.

Another benefit of CDN is that many websites use the same CDN, so the files they use are cached in the user's browser. This can help to decrease the loading time of your website because the files you are using have already been downloaded to the client's browser.

The amount of content stored in a CDN varies among different web applications. Some applications store all their content on a CDN, while other applications store only some of their content.

Microsoft has a dedicated CDN called Microsoft Ajax CDN that hosts some popular libraries, such as jQuery and Bootstrap.



**Note:** Microsoft does not own the license of the libraries but only hosts the libraries for developers.

You can often reduce the loading time of your web application, by using the libraries hosted on Microsoft Ajax CDN. Web browsers can cache these libraries on a local system.

The following code example demonstrates how to link a layout to the jQuery library using a CDN:

### Linking to the jQuery library by using a CDN

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <link href "~/css/site.css" rel="stylesheet" />
 <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"></script>
</head>
<body>
 <div>
 @RenderBody()
 </div>
</body>
```

```
</html>
```

 **Note:** In the preceding code example, the file has the **.min.js** extension, which means this file is minified and is more lightweight than the file with the **.js** extension. Bundling and minification are covered below and in Module 9, "Client-Side Development".

 **Best Practice:** Applications in production should not entirely depend on CDN assets because they might be unavailable. Applications should test the CDN asset referenced and when it is not available, to use a fallback method such as serving a local file.

## Using Package Managers

You can avoid adding libraries manually to your web application by using a package manager. Package manager is a tool that automates the process of installing, upgrading and configuring packages. It also allows developers to create, share and use handy code. Using a package manager helps to reduce the need for configuration tasks while adding libraries to an application. There are several package managers that you can choose from, which include:

- NuGet.
- Yarn.
- Webpack.
- Bower.
- npm.

### NuGet

NuGet is a free and open source package manager which Microsoft has created for the Microsoft development platform and it is part of Visual Studio. In past versions of ASP.NET, it was recommended to use the NuGet package manager to install client-side libraries such as jQuery and Bootstrap. In ASP.NET Core this has changed and the NuGet package manager should be used mainly to install server-side packages.

### Yarn

Yarn is a very fast and secure package and dependency manager. It allows to use and share code with other developers. Yarn checks the integrity of the packages being installed before executing the code.

### Webpack

Webpack is a module bundler for JavaScript applications and serves as a task runner. It processes your application and creates a dependency graph. This graph maps exactly what are the modules that your application needs. Webpack creates bundles including the project's dependencies.

### Bower

Bower is a lightweight package manager that helps to manage frameworks, libraries, and assets on the web. It does not bundle the packages or perform tasks as Webpack but only installs the packages your application needs.

### npm (Node Package Manager)

One of the most popular package managers that allows adding libraries and front-end frameworks to your application. It has many registered packages and the number is constantly growing.



**Note:** In this course npm will be used to install and manage client-side libraries.

## Using npm in ASP.NET Core Applications

You can use npm in your application by using the npm CLI. The npm CLI is available in the terminal. You can access the npm CLI in the Visual Studio 2017 in the Package Manager Console. To install a package registered in the npm registry, you can use the **npm install** command.



**Additional Reading:** You can learn more about npm from the npm's official website:  
<https://aka.ms/moc-20486d-m8-pg3>

To start using npm in your ASP.NET Core application, you should add a **package.json** file to your solution in your project's root folder. This file gives npm information which allows it to identify the project and to handle the project's dependencies. It can also contain more information about the project such as a project description and the version number in a specific distribution.

The **package.json** file includes several kinds of properties. It includes information describing your application which includes application version, name, and privacy settings. The most important properties for developers are **dependencies** and **devDependencies**.

- **dependencies**. This is a list of packages that your client-side logic depends on such as jQuery and Bootstrap. When a package is located in the **dependencies** section, npm will always install it as part of the application.
- **devDependencies**. This is a list of packages that will be installed only in a development environment. Those packages are not related to the client-side logic but to bundling, minification, and compilation among others.

Each package listed in the **dependencies** and **devDependencies** section has a version number that indicates which version of the package will be installed. Visual Studio will always recommend installing the latest stable version of each package.

The following code example demonstrates how to add jQuery using the **package.json** file:

### Adding jQuery by using npm

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "dependencies": {
 "jquery": "3.3.1"
 },
 "devDependencies": {
 }
}
```

After adding jQuery inside the **dependencies** section and saving the **package.json** file, the jQuery package will be downloaded and placed inside the **node\_modules** folder. This folder is located inside the root folder of the application and will contain all the code the application needs.

By default, static files in an ASP.NET Core application can be served only if they are located in the **wwwroot** folder. If, however you want to allow static files to be served from a different folder, for example, the **node\_modules** folder, you can use the **UseStaticFiles** method and pass an instance of type **StaticFileOptions** to it as a parameter.

The following code example demonstrates how to allow static files to be served from the **node\_modules** folder:

#### Add the **node\_modules** folder to the static files middleware inside the **Startup.cs** file

```
public class Startup
{
 public void ConfigureServices(IServiceCollection services)
 {
 services.AddMvc();
 }

 public void Configure(IApplicationBuilder app, IHostingEnvironment env)
 {
 app.UseStaticFiles();
 app.UseStaticFiles(new StaticFileOptions
 {
 FileProvider = new PhysicalFileProvider(
 Path.Combine(env.ContentRootPath, "node_modules")
),
 RequestPath = "/node_modules"
 });
 app.UseMvc();
 }
}
```

The following code example demonstrates how to link the scripts installed by using npm to a layout:

#### Using scripts installed with **npm**

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <link href="~/css/site.css" rel="stylesheet" />
 <script src="..../node_modules/jquery/dist/jquery.js"></script>
</head>
<body>
 <div>
 @RenderBody()
 </div>
</body>
</html>
```

 **Best Practice:** Directly referencing the **node\_modules** folder from inside of views is not considered a good practice. A better practice is to use bundling and minification.

## Bundling and Minification

Script libraries that you add to an application can slow down the loading time of webpages because it takes time to download the scripts into the browser. Bundling and minification help reduce the loading time of web applications by reducing both the number and size of HTTP requests.

- Bundling helps combine multiple JavaScript libraries into a single HTTP request.
- Minification compresses code files before incorporating them in the client application.

To use bundling and minification in an ASP.NET Core application, you can use the **bundleconfig.json** configuration file. The **bundleconfig.json** configuration file defines how the application's static files will be bundled and minified.

The following code shows a **bundleconfig.json** file that adds jQuery to the site bundle:

#### Adding jQuery to a bundleconfig.json File

```
[
 {
 "outputFileName": "wwwroot/css/site.min.css",
 // An array of relative input css file paths.
 "inputFiles": [
 "wwwroot/css/site.css"
]
 },
 {
 "outputFileName": "wwwroot/js/scripts.min.js",
 "inputFiles": [
 // An array of relative input js file paths including jQuery.
 "wwwroot/lib/jquery/dist/jquery.js",
 "wwwroot/js/site.js"
],
 // Optionally specify minification options
 "minify": {
 "enabled": true,
 "renameLocals": true
 },
 // Optionally generate .map file
 "sourceMap": false
 }
]
```

To enable the execution of bundling and minification during the build time, you can download the **BuildBundlerMinifier** NuGet package from the NuGet package manager. This package injects **MSBuild Targets** which are grouped tasks that run at build and clean time.

Each time that you run a Build or Clean in your application, the **bundleconfig.json** file will be analyzed by the build process and the output files will be produced based on the defined configuration.



**Note:** You can learn how to perform bundling and minification by using task runners in ASP.NET Core in Module 9, “Client-Side Development”.

### Demonstration: How to Use npm to Add a Library

In this demonstration, you will learn how to add the jQuery package to an ASP.NET Core application by using npm. You will then see how to add a link from a layout to a jQuery file installed by npm. Since the jQuery file is located in a **node\_modules** folder which is outside the **wwwroot** folder, you will see how to use the **UseStaticFiles** middleware so the jQuery file will be served. Finally, you will see how to add a style sheet file to the application and add a link to it from a layout.

#### Demonstration Steps

You will find the steps in the section “Demonstration: How to Use npm to Add a Library” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD08\\_DEMO.md#demonstration-how-to-use-npm-to-add-a-javascript-library](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_DEMO.md#demonstration-how-to-use-npm-to-add-a-javascript-library).

## Lesson 3

# Using jQuery

jQuery is a JavaScript library that simplifies the adding of JavaScript to HTML pages in general and to views in particular. jQuery is an open-source software that you can use for free. It helps reduce the amount of code that you need to write and perform tasks such as accessing and modifying HTML elements on a webpage.

jQuery contains many selectors. Selectors are used to select and manipulate HTML elements. jQuery also contains several functions. An example of a function in jQuery is the **ajax** function. You can use the **ajax** function in jQuery to call a server.

Validating the user input is very important in web applications. It is possible to validate user input in both the server-side and the client-side. jQuery can be used to help validate user input in the client-side. It is important to know how to do client-side validation by using jQuery.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe jQuery.
- Describe how jQuery helps access the HTML elements of a webpage.
- Describe how to modify elements by using jQuery.
- Describe how to call a server by using jQuery.
- Describe client-side validation by using jQuery.

### Introduction to jQuery

jQuery is a JavaScript library that you can use within different browsers. jQuery was first released in 2006. jQuery helps query the HTML Document Object Model (DOM) and obtain a set of HTML DOM elements. This feature of jQuery helps to:

- Reduce the amount of code that you need to write to perform a task.
- Reduce the client-side development time of web applications.

jQuery includes the following features:

- DOM element selections.
- DOM traversal and modification.
- DOM manipulation, based on CSS selectors.
- Events.
- Effects and animations.
- AJAX.
- Extensibility through plug-ins.

- jQuery is a cross-browser JavaScript library
- Benefits of using jQuery:
  - It reduces the amount of code that you need to write
  - It reduces the application development time
- jQuery files:
  - jQuery original version:
    - Is the uncompressed version of jQuery
    - Is optimized for development and debugging
  - jQuery minified version:
    - Is the compressed version of jQuery
    - Is optimized for production

- Utilities.
- Compatibility methods.
- Multi-browser support.



**Additional Reading:** For more information about jQuery, go to: <https://aka.ms/moc-20486d-m8-pg4>

## jQuery Files

When working with jQuery, you can choose between working with an original version or a minified version. The jQuery original version and jQuery minified version provide similar functionalities; however, they optimize web applications for different purposes:

- *jQuery original version* (`jQuery-<version>.js`). This is the uncompressed version of jQuery.
- *jQuery minified version* (`jQuery-<version>.min.js`). This includes the compressed and gZip versions of jQuery.

When you are in the production environment, you can use the jQuery minified version to reduce the loading time of the web application. However, if you use the minified version while working on the development environment, you cannot access the source code of the JavaScript libraries during the debug operation. Therefore, you can use the original version of jQuery, while you are in the development environment.

In addition to the regular version of jQuery, there is also a slim version of jQuery. The slim version of jQuery excludes modules that exist in the original version of jQuery, such as ajax and effects. It also excludes deprecated code.

## The jQuery Syntax

Every jQuery line of code starts with the `$` or `jQuery` variables and it allows you to call functions that exist inside the jQuery interface, to query the DOM and perform DOM manipulations.

After adding the jQuery library to your application, a global variable is automatically added to the `window` object named `$`. It holds the whole interface which jQuery exposes and it also includes the `query` function that allows it to traverse the DOM. jQuery also adds a global variable named `jQuery` that is pointing to the same object as the `$` variable. This variable is used when another library or code is using the `$` sign and you want to avoid collision between the jQuery library and the other library.

The following code example introduces the jQuery syntax. It shows how to access the `each` function inside a view:

### The jQuery syntax

```
@{
 Layout = "_Layout";
}

<h2>Index</h2>
<script>
 $(function() {
 $.each([4, 9], function(index, value) {
 alert(index + ":" + value);
 });
 });
</script>
```

The **each** function in jQuery is an iterator function. It can be used to iterate an array. The elements in the array are iterated by a numeric index, from 0 to the length of the array minus 1. When the jQuery code of the example above is run in the browser, two popups are shown to the user: the first one has the text **0:4**, and the second has the text **1:9**.

The following code example demonstrates how you can use the **jQuery** variable instead of the **\$** variable:

### Replacing \$ with jQuery

```
@{
 Layout = "_Layout";
}
<h2>Index</h2>
<script>
 jQuery(function() {
 jQuery.each([4, 9], function(index, value) {
 alert(index + ":" + value);
 });
 });
</script>
```

## Accessing HTML Elements by using jQuery

jQuery helps to access and manipulate HTML elements, to create interactive web applications. jQuery selector syntax starts with a call to the **jQuery** object which accepts a string consisting of CSS selectors.

The following example demonstrates how to use jQuery selectors to select sets of elements by using element name, id, or CSS class:

### Using jQuery selectors

```
$(element name|#id|.class)
```

- You can use the following selector to select elements by element name, id, or CSS class:

```
$(element name|#id|.class)
```

- After accessing the HTML elements:
  - Modify the attributes on the elements
  - Define event handlers to respond to events

```
$("#hello").click(function(event) {
 alert("Hello World");
});
```

The following jQuery selector accesses the HTML element with the **hello** id:

### Using a jQuery selector to access an HTML element with hello id

```
$("#hello")
```

You can use jQuery to access all instances of a specific HTML element.

The following jQuery selector identifies all instances of the **a** element:

### Accessing all instances of the a element

```
 $("a")
```

You can use more than one selector in a single query to access multiple types of elements at once.

The following code example demonstrates how to identify all instances of the **a** element, all instances of the **div** element which have the **big** class, and an element with **hello** id, and put them together in a single jQuery array:

### Using more than one selector in a query

```
$("a, div.big, #hello")
```

After accessing the HTML elements, you can perform actions on the elements, such as:

- Defining event handlers to respond to events associated with the selected HTML elements.
- Reading the attributes of the HTML elements.

The following example adds an event handler to the **click** event of an HTML element with **hello** id:

### Using a jQuery event handler

```
$("#hello").click(function(event) {
 alert("Hello World");
});
```

## The document.ready() and \$() Functions

If the jQuery scripts load before the webpage loads, you might encounter errors such as **object not defined**. You can place the jQuery code in the **document.ready** function, to prevent the code from loading until all HTML elements in the page load.

The following code shows how to use the **document.ready** function:

### Using the document.ready function

```
$(document).ready(function() {
 // Code placed here will not execute before all HTML elements in the page load.
});
```

Experienced developers usually shorten this code by using the **\$()** syntax instead of **\$(document).ready**.

The following example demonstrates how to use the **\$()** syntax instead of **\$(document).ready()**:

### Shorthand for \$(document).ready

```
$(function() {
 // Code placed here will not execute before all HTML elements in the page load.
});
```

The following example shows how to access HTML elements by using jQuery inside a view:

### Using jQuery in a view

```
@{
 ViewData["Title"] = "jQuery Demo";
}
<h2>@ViewData["Title"]</h2>

<div>
 Hello
 <input type="button" value="Hello" id="hello" />
</div>
<script>
 $(function() {
 $("#hello").click(function(event) {
 alert("Hello World");
 });
 });
```

```
});
</script>
```



**Best Practice:** It is important to load jQuery library to the page before calling the `$()` or `$(document).ready()` functions. Using the `$` object before the jQuery library is loaded might cause an exception. When using a layout file to load scripts, place the jQuery library source script inside the `<head>` tag before all the other files.

## Use jQuery to Read Attributes of HTML Elements

You can use functions in jQuery to read the attributes of the HTML elements. An example of such a function is the `text` function. You can use the `text` function to get the text inside of HTML elements. If there are multiple HTML elements inside a specific element, the function will return the combined text of all HTML elements.

The following example shows how to get the text inside of a `h1` element:

### The `text` function

```
<h1>This is the main headline of a HTML document</h1>

<script>
$(function() {
 let res = $('h1').text();
 console.log(res); // Result: "This is the main headline of a HTML document".
});
</script>
```

## Modifying HTML Elements by using jQuery

You can use jQuery to query HTML DOM and obtain HTML elements. You can use jQuery functions to modify the attributes associated with the HTML elements. The following are some commonly used jQuery functions, which enable you to modify HTML elements.

### The `val` Function

You can use the `val` function to get or set the value of an HTML element.

The following example shows how to use the `val` function to set the value of the element with the `hello` id to "Hello World":

### The `val` function

```
$("#hello").val('Hello World');
```

jQuery functions include:

- The `val` function:
  - Allows to get or set the value of an HTML element
- The `css` function:
  - Allows to get or set the inline CSS style associated with an HTML element
- The `addClass` function:
  - Assigns the CSS class to an HTML element
- The `animate` function:
  - Creates transition between CSS properties assigned to an HTML element

### The `css` Function

You can use the `css` function to get or set the inline CSS style associated with an HTML element.

The following example shows how to use the `css` function to set the background color of an HTML element to blue:

### The **css** function

```
$('#hello').css('background-color', 'blue');
```

### The **addClass** Function

You can use the **addClass** function to assign a CSS class to an HTML element.

The following example shows how to use the **addClass** function to add the **input\_css\_class** to an HTML element:

### The **addClass** function

```
$('#hello').addClass('input_css_class');
```

### The **animate** Function

You can use the **animate** function to animate CSS properties of HTML elements.

The following example shows how to use the **animate** function to create a transition of the **fontSize** and **letterSpacing** properties of an HTML element. The animation will run over a period of 1200 milliseconds:

### The **animate** function

```
$("#hello").animate({
 "fontSize": "32px",
 "letterSpacing": "10px"
}, 1200);
```



**Additional Reading:** For more information about jQuery functions that modify HTML elements, go to <https://aka.ms/moc-20486d-m8-pg1>

## Demonstration: How to Modify HTML Elements by using jQuery

In this demonstration, you will see how to use jQuery to read data from an HTML element by using the **text** function. You will also see how to modify an HTML element by using the **addClass** function.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Modify HTML Elements by Using jQuery” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD08\\_DEMO.md#lesson-3-using-jquery](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_DEMO.md#lesson-3-using-jquery).

## Calling a Server by using jQuery

jQuery includes the **ajax** function that helps:

- Perform asynchronous calls to a server.
- Retrieve the data returned from a server.

You can pass several parameters to the **ajax** function to control how to call the server. The following are examples for parameters that can be passed to the **ajax** function:

- **method**. This parameter controls the request method that you can use while calling the server. Common method values are: GET, POST, PUT and DELETE.
- **dataType**. This parameter defines the data type you expect to get from the server.
- **url**. This parameter provides the URL of the server.
- **data**. This parameter defines the data that you should provide as a parameter to the server.
- **contentType**. This parameter defines the HTTP content type that you should use, when you submit HTTP requests to the server.

### The **ajax** function:

- Helps perform calls to a server
  - Includes parameters such as **type**, **url**, **data**, and **contentType**
- Helps obtain the data returned from a server
  - Uses callback functions to derive the results of the server calls

The **ajax** function returns an object that implements the Promise interface. The object returned from the **ajax** function has several functions, which include:

- *done*. This function is triggered when the call completes successfully.
- *fail*. This function is triggered when the call completes with errors.

When a call to a server completes, jQuery triggers one of the two callback functions based on the success of the call.

The following example illustrates how to use the **ajax** function:

### Using the **ajax** function

```
$.ajax({
 method: "GET",
 dataType: "json",
 url: "http://server-domain/api/Customer",
 data: {'id': '123'},
 contentType: "application/json; charset=utf-8"
}).done(function(msg) {
 alert("Data received: " + msg);
}).fail(function (msg) {
 alert(msg);
});
```

 **Note:** In this topic, only the client-side code is demonstrated. The server-side code doesn't appear in this topic. An example of a server is Web API. Web API is covered in detail in Module 13, "Implementing Web APIs".

 **Additional Reading:** For more information about the **ajax** function, go to <https://aka.ms/moc-20486d-m8-pg2>

## Client-Side Validation by using jQuery

Before saving any data to a database, it is important to perform data validations to ensure the validity of the information that needs to be stored. This is especially true when you want to store data submitted by users. You must be sure that there are no potential security threats, verify that it is formatted as you expect it to be and that it conforms to your type and size rules.

In ASP.NET Core MVC applications, data validation can happen both on the client-side and the server-side. You could potentially leave the validation to the server-side, but server-side validation consumes more time. Server-side validation alone is inconvenient to the users because they will have to wait until the whole round trip to the server is complete in order to know whether their input is correct or not. Even though it might seem to be only a few fractions of a second, it adds up to be a lot of time and causes frustration.

On the other hand, client-side validation alone is not enough. Any advanced user can easily bypass it and send any information they want to the server. That is why it is a best practice to have both client-side validation and server-side validation.

In Module 6, "Developing Models", you learned how to apply a server-side validation in ASP.NET Core MVC applications. In this topic, you will see how to apply a client-side validation in ASP.NET Core MVC applications.

### Adding Client-Side Validations

To perform client-side validation in an ASP.NET Core MVC application, you need to add links to two JavaScript scripts, along with the jQuery library you saw earlier in this lesson:

- **jQuery Validate.** A jQuery plugin that makes client-side form validations easy. The plugin has a built-in bundle of useful validation methods, including URL and email validation. You can perform these validations before a form submit or after any user interaction.
- **jQuery Unobtrusive Validation.** A custom Microsoft library that is built based on the jQuery Validate plugin. This plugin allows you to use the same validation logic you wrote on the server-side in the form of data annotations and metadata and apply it to the client-side immediately.

The following code example demonstrates a **package.json** file which includes jQuery Validate and jQuery Unobtrusive Validation:

### Adding jQuery validation scripts to package.json

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "devDependencies": {},
 "dependencies": {
 "jquery": "3.3.1",
 "jquery-validation": "1.17.0",
 "jquery-validation-unobtrusive": "3.2.10"
 }
}
```

- Advantages of client-side validation:

- Immediate validation
- No need to wait for a server response
- Better user experience

- Disadvantages of client-side validation:

- Less secure
- Easy to bypass

- The best practice in MVC applications is to have both server and client validation

The following code example demonstrates how to link the scripts installed by using npm to a layout:

### Using scripts iwith npm

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>@ViewBag.Title</title>
 <link href("~/css/site.css" rel="stylesheet" />
 <script src="../node_modules/jquery/dist/jquery.js"></script>
 <script src "~/node_modules/jquery-validation/dist/jquery.validate.min.js"></script>
 <script src "~/node_modules/jquery-validation-
unobtrusive/dist/jquery.validate.unobtrusiv e.min.js"></script>
</head>
<body>
 <div>
 @RenderBody()
 </div>
</body>
</html>
```

When using jQuery Unobtrusive Validation, instead of writing validation logic in two places, you will write it once. MVC's Tag Helpers and HTML Helpers consume the validation attributes and type metadata from model properties and render HTML 5 attributes starting with **data-** on form elements needing validation. MVC will generate the **data-** attributes for both built-in and custom attributes.

jQuery Unobtrusive Validation will then parse the **data-** attributes and passes the logic to jQuery Validate. This way you can duplicate the server-side validation logic to the client without writing the same logic twice.

The following code example demonstrates a model with validation attributes:

### A model with validation attributes

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

 [DataType(DataType.EmailAddress)]
 [Required(ErrorMessage = "Please enter your email address")]
 public string Email { get; set; }
}
```

The following code example demonstrates how to use jQuery Unobtrusive Validation to implement client-side validation:

### Client-side validation by using jQuery unobtrusive validation

```
@model Customer

<form method="post" asp-action="SomeAction">
 <div class="form-field">
 <label asp-for="Name"></label>
 <input asp-for="Name" />

 </div>
 <div class="form-field">
 <label asp-for="Email"></label>
 <input asp-for="Email" />

 </div>
 <div class="form-field">
 <input class="submit-btn" type="submit" />
 </div>
</form>
```

```
</div>
</form>
```

# Lab: Using Layouts, CSS and JavaScript in ASP.NET Core MVC

## Scenario

You have been asked to add a slideshow to the homepage of the zoo web application that will show some of the animals' photos. 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 order.

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 have been also asked to add a purchase page, to enable customers to buy adult, child, and senior tickets to the zoo. You will use jQuery to do calculations on the page. You will also use client-side validation to validate the input typed by the users.

## Objectives

After completing this lab, you will be able to:

- Apply a consistent look and feel to the web application.
- Use layouts to ensure that common interface features, such as the headers, are consistent across the entire web application.
- Render and execute JavaScript code in the browser.
- Use the jQuery script library to update and animate page elements.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD08\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD08\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_LAK.md).

## Exercise 1: Applying a Layout and Link Views to it

### Scenario

To construct a web application with a consistent look and feel, a layout should be added to the web application. In this exercise, you will create a layout and link views to it.

The main tasks for this exercise are as follows:

1. Create a layout.
2. Add a view and link it to the layout.
3. Add \_ViewStart.cshtml.
4. Add existing views to the web application.
5. Add a section to the layout.
6. Run the application.

## Exercise 2: Using CSS

### Scenario

To improve the appearance of the web application, a CSS should be used. In this exercise, you will add a CSS file to the web application and add a link from the layout to the CSS file.

The main tasks for this exercise are as follows:

1. Add an existing CSS file to the project.
2. Link the layout to the CSS file.
3. Style the menu.
4. Style the photos section in Index.cshtml.
5. Style a form in BuyTickets.cshtml.
6. Run the application.

## Exercise 3: Using JavaScript

### Scenario

To calculate the total cost of the tickets, you have been asked to add a function in JavaScript. In this exercise, you will add a JavaScript file and add a link to the JavaScript file from a view.

The main tasks for this exercise are as follows:

1. Add a JavaScript file.
2. Link a view to the JavaScript file.
3. Write the code of the JavaScript file.

## Exercise 4: Using jQuery

### Scenario

You have been asked to handle click events, modify elements, and change the style of elements. You are also asked to apply client-side validation in the web application. In this exercise, you will use npm to add several client-side packages to the web application and you will use the packages to make various operations in the client-side.

The main tasks for this exercise are as follows:

1. Use npm to add jQuery.
2. Use jQuery to add event handlers.
3. Use jQuery to modify elements.
4. Use jQuery for Client-side validation.
5. Run the application.

**Question:** How did you apply the same layout to all the views in the web application?

**Question:** Your development team has decided to use a new client-side package in the zoo web application. You were asked by your manager to add the client-side package in a similar way to the client-side packages that already exist in the zoo web application. How would you add the client-side package?

# Module Review and Takeaways

In this module, you learned about the key role that layouts play in the construction of an MVC web application and how they can be used to apply a consistent look and feel to a web application. You also learned how to use basic CSS and JavaScript abilities and how to link your layout to external CSS and JavaScript files. Then you learned about the various ways to add external JavaScript libraries to your application and particularly how to add jQuery to your views. Lastly, you learned how to use jQuery to manipulate the DOM, change HTML elements' attributes, apply CSS rules and perform client-side validation.

## Review Question

**Question:** Why do you have sections in a layout?

## Best Practices

- Use npm to add client-side libraries to your application. Avoid installing client-side libraries by using the NuGet packages manager.
- When using a CDN to add scripts to your application, create a fallback in case the script is not available on the CDN.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You get an error that indicates that a script you want to use is not loaded to your application.	



# Module 9

## Client-Side Development

### Contents:

Module Overview	9-1
<b>Lesson 1:</b> Applying Styles	9-2
<b>Lesson 2:</b> Using Task Runners	9-19
<b>Lesson 3:</b> Responsive Design	9-33
<b>Lab:</b> Client-Side Development	9-42
Module Review and Takeaways	9-44

## Module Overview

When creating an application, it is important to know how to develop both client-side and server-side of the application. In this module, you are going to learn client-side tools that will allow you to create extensive web applications on any scale. These tools are based on the topics covered in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

In this module, you are going to learn how to use the Bootstrap framework to style your web application. Then you are going to learn how to use Sass and Less, two common Cascading Style Sheets (CSS) preprocessors that add features to CSS, such as variables, nested rules, and functions. These greatly improve the maintainability of complex web applications.

Next, you will learn how to set up task runners such as Grunt and gulp and how to use them to compile Sass files during the Microsoft Visual Studio build. You will learn how to use the gulp task runner to perform bundling and minification of CSS and JavaScript files and how to set up a watcher task to compile Sass files as you write your code, without the need to rebuild the solution.

Finally, you will learn responsive design tools that allow you to customize your web application's display based on the capabilities and specifications of a web browser or a device. You will learn to write CSS media queries, how to use the Bootstrap responsive grid system, and how to apply the CSS flexbox layout to your views.

### Objectives

After completing this module, you will be able to:

- Use Bootstrap, Sass and Less in a Microsoft ASP.NET Core application.
- Use task runners in an ASP.NET Core application.
- Ensure that a web application displays correctly on devices with different screen sizes.

# Lesson 1

## Applying Styles

To style your web application, you need to write lots of CSS code. There are many tools that allow you to speed up the process of writing and applying styles to web applications. In this lesson, you are going to learn some of these tools.

You are going to learn how to use Bootstrap, a framework that includes many pre-built components and styles that can be applied to your applications. Also, you are going to learn how to use Sass and Less, CSS preprocessors, that are compiled to CSS and can help you to style your web application.

### Lesson Objectives

After completing this lesson, you will be able to:

- Add Bootstrap to your web application.
- Use various Bootstrap components.
- Apply styles to your web applications by using Sass.
- Apply styles to your web applications by using Less.

### Introduction to Bootstrap

Bootstrap is an open-source framework for building responsive web applications using HTML, CSS, and JavaScript. It allows you to quickly develop the client-side of your application by using pre-built components and styles.

 **Note:** This course uses Bootstrap v4.1.3. If you choose to use another version of Bootstrap, the syntax might change. When using MVC pre-made templates, the Bootstrap version might also be different.

#### What is Bootstrap?

- Bootstrap is a HTML, CSS and JS framework for building responsive web applications
- Allows you to quickly develop the client-side of your web applications
- Includes pre-built components and styles

### Adding Bootstrap to Your Project

To add Bootstrap to your application, you should add the following dependencies:

- **jQuery**. Bootstrap is based on jQuery. To use Bootstrap components, you should include jQuery in your application.
- **popper.js**. Another Bootstrap dependency. This script library allows you to manage elements that "pop out" of the natural flow of your application.

 **Note:** jQuery is covered in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

After adding the dependencies, you should also include **bootstrap.css**, which includes the styles and **bootstrap.js**, which includes the scripts. If you are using the production environment, include the minified versions.

There are many ways to add Bootstrap to your application. In this topic, you will see how you can use npm to add Bootstrap to your web application.

 **Note:** Various ways to add external client-side libraries to ASP.NET Core applications are covered in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

The following code example demonstrates how to add Bootstrap and its dependencies to the **package.json** file:

### Adding Bootstrap by using npm

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "dependencies": {
 "bootstrap": "4.1.3",
 "jquery": "3.3.1",
 "popper.js": "1.14.3"
 },
 "devDependencies": {
 }
}
```

After adding **package.json** to your web application, the Bootstrap framework and its dependencies will be downloaded to the **node\_modules** folder.

The following example demonstrates how to link a layout to Bootstrap and its dependencies:

### Link a Layout to Bootstrap

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <link href="~/node_modules/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
 <title>@ViewBag.Title</title>
</head>
<body>
 <div>
 @RenderBody()
 </div>

 <script src="~/node_modules/jquery/dist/jquery.js"></script>
 <script src="~/node_modules/popper.js/dist/umd/popper.js"></script>
 <script src="~/node_modules/bootstrap/dist/js/bootstrap.js"></script>
</body>
</html>
```

 **Additional Reading:** For full documentation of Bootstrap, go to the following link:  
<https://aka.ms/moc-20486d-m9-pg3>

## Bootstrap Components

Bootstrap includes many components that can be added to your web application. Some of these components are composed of pure HTML and CSS markup and some components need JavaScript code to work. In this topic, you will see several common components that you can use in your application.

To make Bootstrap components work, it is important to remember that each component has a very specific HTML structure and a corresponding CSS code. If you create an HTML structure that is different from the one that Bootstrap expects, your website might not look as you wanted.

- Bootstrap includes a large variety of pre-built components, such as:  
Alerts, Buttons, Dropdowns, Nav menus, Navbars, Modal windows and more
- To create these components, it is important to follow the right HTML hierarchy that is described inside the Bootstrap documentation and add all needed CSS classes
- When a required class is missing or the hierarchy is not right, the elements will not look and behave as expected

### Alerts

Alerts are used to display messages to users as a result of their actions. Bootstrap alerts are responsive and can include links inside them.

To create an alert, you can add a **div** to your view and apply two CSS classes to this div. The first CSS class is **alert**, which defines this div as an alert. The second class that needs to be added will define the alert type and color. For instance, the **alert-danger** class defines the alert with a red color scheme. Other alert classes include **alert-primary**, **alert-secondary**, **alert-success**, **alert-warning**, **alert-info**, **alert-light** and **alert-dark**. You can add a **role="alert"** attribute to the **div** to appropriately convey its purpose to assistive technologies such as screen readers. To create a link inside an alert, add an **<a>** tag and give it a CSS class of **alert-link**.

The following code example demonstrates how to add designed alert messages to a view:

### Using Bootstrap Alerts

```
@{
 ViewBag.Title = "Bootstrap Alerts Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows designed alert messages</h3>

<div class="alert alert-danger" role="alert">
 This is a designed alert message
</div>
<div class="alert alert-success" role="alert">
 This is an alert with link inside
</div>
```



**Additional Reading:** Full documentation of Bootstrap alerts is located at:  
<https://aka.ms/moc-20486d-m9-pg4>

### Buttons

Buttons are used to create styled action elements inside forms, alerts, tables, grids and more. Bootstrap includes several predefined styles for buttons with multiple size options and states.

To create a button, you can add a **button** HTML element to your view and apply CSS classes to it. A mandatory CSS class is **btn**, which applies Bootstrap design for buttons to this element. You can then apply several CSS classes that define the button characteristics. These CSS classes include:

- Button color classes: btn-primary, btn-secondary, btn-success, btn-danger, btn-warning, btn-info, btn-light, btn-dark.
- Button with outline: btn-outline-primary, btn-outline-secondary, btn-outline-success, btn-outline-danger, btn-outline-warning, btn-outline-info, btn-outline-light, btn-outline-dark.
- Button size classes: **btn-lg**, **btn-sm**.
- Button state classes: **active**, **disabled**.

The following code example demonstrates how to add designed buttons to a view:

### Using Bootstrap Buttons

```
@{
 ViewBag.Title = "Bootstrap Buttons Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows Bootstrap designed buttons</h3>

<button type="button" class="btn btn-outline-primary">Primary Style</button>
<button type="button" class="btn btn-outline-secondary">Secondary Style</button>
<button type="button" class="btn btn-success btn-lg">Large Success button</button>
<button type="button" class="btn btn-danger btn-sm">Small Danger button</button>
<button type="button" class="btn btn-outline-info">Info button</button>
<button type="button" class="btn btn-outline-light">Light button</button>
```

 **Additional Reading:** Full documentation of Bootstrap buttons is located at:  
<https://aka.ms/moc-20486d-m9-pg5>

### Dropdowns

Dropdowns are used to create lists of links that overlay the content of the page when they are shown. Dropdowns are togglable and interactive.

To create a dropdown list, you can add a div with the **dropdown** class, which will include all the elements needed to be displayed. Then, you can add a button or link that when clicked will toggle the display of the dropdown list. This button can have all the classes that were covered above in the description of Bootstrap buttons. Also, this button needs to have the **dropdown-toggle** class and the **data-toggle="dropdown"** attribute that will allow it to toggle the display of the dropdown list.

After the button, you need to add a div with the **dropdown-menu** class. This div will include all the links and items displayed inside the dropdown list. An item inside the dropdown list should have the **dropdown-item** class. To separate items in a dropdown list, you can add a div with the **dropdown-divider** class.

 **Note:** Make sure that you include the **popper.js** script before using dropdowns because Bootstrap dropdowns are built on top of this plugin. Adding **popper.js** to ASP.NET Core application is covered in the previous topic, "Introduction to Bootstrap".

The following code example demonstrates how to add a dropdown to a view:

### Using Bootstrap Dropdown

```
@{
 ViewBag.Title = "Bootstrap Dropdown Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap designed dropdown</h3>
```

```
<div class="dropdown">
 <button type="button" class="btn btn-info dropdown-toggle" data-toggle="dropdown">
 Info Button with Dropdown
 </button>
 <div class="dropdown-menu">
 Link A
 Link B
 Link C
 <div class="dropdown-divider"></div>
 Separated Link
 </div>
</div>
```



**Additional Reading:** Full documentation of Bootstrap dropdowns is located at:

<https://aka.ms/moc-20486d-m9-pg6>

## Forms

Bootstrap includes styles for many types of form controls that can be used practically in any web application. Bootstrap form controls include: textual form controls such as `<input type="text">` and `<textarea>`, password field, checkbox, radio button, select, file input, range input and more.

To get the desired look, it is important to follow the required hierarchy of the HTML elements and the CSS classes. For example, to create an input element for some text field that has a label, the label and input HTML elements should be under a div with the **form-group** class. Each input control should have the **form-control** class.

The following code example demonstrates how to add bootstrap form controls to a view:

### Using Bootstrap Form Controls

```
@{
 ViewBag.Title = "Bootstrap Form Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap designed form</h3>

<form>
 <div class="form-group">
 <label for="firstname">First Name</label>
 <input type="text" class="form-control" id="firstname" placeholder="First Name">
 </div>
 <div class="form-group">
 <label for="lastname">Last Name</label>
 <input type="text" class="form-control" id="lastname" placeholder="Last Name">
 </div>
 <div class="form-group">
 <label for="email">Email</label>
 <input type="email" class="form-control" id="email" placeholder="Email">
 </div>
 <div class="form-group">
 <label for="password">Password</label>
 <input type="password" class="form-control" id="password" placeholder="Password">
 </div>
 <div class="form-group">
 <label for="city">City</label>
 <input type="text" class="form-control" id="city">
 </div>
 <div class="form-group">
 <label for="state">State</label>
 <select id="state" class="form-control">
 <option selected>Choose...</option>
 <option>NY</option>
 </select>
 </div>
</form>
```

```

 <option>..</option>
 </select>
</div>
<div class="form-group">
 <label for="zip">Zip</label>
 <input type="text" class="form-control" id="zip">
</div>
<button type="submit" class="btn btn-primary">Sign Me Up</button>
</form>

```



**Additional Reading:** Full documentation of Bootstrap forms is located here:  
<https://aka.ms/moc-20486d-m9-pg7>

## Navs

Bootstrap has a large variety of responsive and mobile friendly navigational controls. They can allow you to add navigation menus easily. All navigational controls share the **nav** class. Each item in the nav menu should have the **nav-item** class. Each link inside the menu item should have the **nav-link** class. The **disabled** class should be used on disabled links and the **active** class should be used on the currently active link.

The following code example demonstrates how to add a Bootstrap nav to a view:

### Using Bootstrap Nav

```

@{
 ViewBag.Title = "Bootstrap Nav Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap designed nav</h3>

<ul class="nav">
 <li class="nav-item">
 Home Page

 <li class="nav-item">
 About

 <li class="nav-item">
 Contact

 <li class="nav-item">
 Disabled Link


```



**Additional Reading:** Full documentation of Bootstrap navs is located at:  
<https://aka.ms/moc-20486d-m9-pg8>

## Navbars

A navbar is a Bootstrap responsive navigation header that can include the website name, menu and more. To get started with navbars, it is important to be familiar with the following:

- Each navbar should be wrapped with at least two classes: **navbar** and a class compound of **navbar-expand** and one of the following endings: **-sm**, **-md**, **-lg**, or **-xl**. For example, **navbar-expand-sm** or **navbar-expand-md** and so on.
- Each navbar wrapper can have a class that defines its color scheme, for example, **navbar-light** or **navbar-dark**. You can also customize the color schema with **bg-\*** classes, for example, **bg-light**.

- Navbar comes with pre-built sub-components which include:
  - **navbar-brand**. Can be used to display the name of the company or website.
  - **navbar-nav**. Can be used for navigation.
  - **navbar-toggler**. Can be used for toggling. You should also add the **data-toggle="collapse"** attribute and the **data-target="#id"** attribute, where **id** stands for the element needed to be toggled. In case you want the icon to look like a hamburger, you can use the **navbar-toggler-icon** class.
  - **collapse** and **navbar-collapse**. Can be used to group and hide navbar contents.

The following code example demonstrates how to add a Bootstrap navbar to a view:

### Using Bootstrap Navbar

```
@{
 ViewBag.Title = "Bootstrap Navbar Example";
 ViewBag.CompanyName = "Contoso";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap designed navbar</h3>

<nav class="navbar navbar-expand-lg navbar-light bg-light">
 @ViewBag.CompanyName
 <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav">

 </button>
 <div class="collapse navbar-collapse" id="navbarNav">
 <ul class="navbar-nav">
 <li class="nav-item active">
 Home

 <li class="nav-item">
 About

 <li class="nav-item">
 Contact

 </div>
</nav>
```



**Additional Reading:** Full documentation of Bootstrap navbars is located at:

<https://aka.ms/moc-20486d-m9-pg9>

### Modals

Bootstrap modal windows are used to display dialog messages and user notifications inside your application. They are placed over the site content and can be closed by clicking on the backdrop of the modal. Only one modal window can be shown at a time.

To trigger a modal window, you can apply two attributes to a button. The first attribute is **data-toggle="modal"**, which opens the modal window. The second attribute is **data-target="#id"** where **id** stands for the id of the modal.

The modal can be a div that has a **modal** class. To improve accessibility for people that use screen readers you can apply the **role="dialog"** attribute to the div. You need to use the **modal-dialog** class to properly set the width and margin of the modal properly.

To set the styles of the content of the modal, you need to use the **model-content** class. The modal content can have several parts which include:

- **modal-header**. Can be used to style the header of the modal.
- **modal-body**. Can be used to style the body of the modal.
- **modal-footer**. Can be used to style the footer of the modal.

The following code example demonstrates how to add a Bootstrap modal to a view:

### Using Bootstrap Modal

```
@{
 ViewBag.Title = "Bootstrap Modal Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap designed modal</h3>

<button type="button" class="btn btn-danger" data-toggle="modal" data-
target="#modalWindow">
 Click to Open a Modal Window
</button>

<div id="modalWindow" class="modal" role="dialog">
 <div class="modal-dialog" role="document">
 <div class="modal-content">
 <div class="modal-body">
 <p>This is a modal window</p>
 </div>
 <div class="modal-footer">
 <button type="button" class="btn btn-primary" data-
dismiss="modal">OK</button>
 </div>
 </div>
 </div>
</div>
```

 **Additional Reading:** Full documentation of Bootstrap modals is located at:  
<https://aka.ms/moc-20486d-m9-pg10>

## Demonstration: How to Work with Bootstrap

In this demonstration, you will first learn how to use npm to add Bootstrap and its dependencies to an ASP.NET Core application. After that, you will learn how to add Bootstrap components to the application.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Work with Bootstrap” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD09\\_DEMO.md#demonstration-how-to-work-with-bootstrap](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD09_DEMO.md#demonstration-how-to-work-with-bootstrap).

## Styling Applications with Sass

There is an expectation today that websites should look good and provide an excellent user experience. This expectation has led developers to write an extensive amount of CSS. A need was born to find a method to make CSS easier to define and maintain. This is how the CSS preprocessors were born.

A preprocessor is a language that is compiled to another language. The goal of a preprocessor is to improve the experience of working with the underlying language.

A CSS preprocessor is a language that is compiled to CSS. Sass and Less are examples of CSS preprocessors. They allow you to add features to CSS, such as variables and functions which improve the maintainability of large and complex applications.

By adding these features, you allow CSS to be more like a real programming language and this can help you to reduce the duplication of code and provide a better organization of your styles.

- Sass is a language that is compiled to CSS

- Installing Sass by using npm:

```
npm install -g sass
```

- Compiling Sass file:

```
sass main.scss main.css
```

 **Note:** Both Sass and Less are CSS preprocessors providing similar features with some unique extensions. Less will be covered in the next topic "Styling Applications with Less".

### Using Sass in your application

You can add a Sass file to your application by adding a file ending with **.scss** extension to your project. You will then need to compile the Sass file to a CSS file.

 **Best Practice:** Place Sass files inside of a dedicated Styles folder which is not located inside the **wwwroot** folder. Place the compiled CSS files inside the **wwwroot** folder.

To compile a Sass file, you can use several techniques, which include:

- Install Sass globally by using npm and then using the **sass** command. This technique will be covered in this topic.
- Configure a task runner. This technique will be covered in Lesson 2, "Using Task Runners".

To install Sass globally, you can run the following command:

#### Installing Sass

```
npm install -g sass
```

The following code example demonstrates how to compile the **main.scss** file to the **main.css** file:

#### Compiling a Sass File

```
sass main.scss main.css
```

## Variables

Variables are one of the basic features Sass adds to CSS. Variables are a fundamental part of any programming language and now you can use them with CSS. Sass variables are often used to define colors, fonts, dimensions and any values that are repeated multiple times throughout the application. This allows you to define theme variables once and then inject them where needed.

Variables help you to create maintainable CSS. When you need to change the general theme of your website, you will mainly need to change the values of your variables without changing the rest of the code.



**Note:** Variables are defined using the \$ symbol.

The following code example demonstrates how to define and use Sass variables:

### Using Sass Variables

```
$highlights : #00FFFF;

.message {
 color: $highlights;
 font-size: 12px;
}

h1 {
 color: $highlights;
}
```

The following code example shows the CSS output you will get after Sass compilation:

### Sass Variables After Compilation

```
.message {
 color: #00FFFF;
 font-size: 12px;
}

h1 {
 color: #00FFFF;
}
```

## Nesting Styles

CSS doesn't support nested hierarchy of selectors. When working with Sass, you can define styles by using nested selectors and use a syntax that is more natural and easier to read.

The following code example demonstrates how to define the style of elements by using nested selectors:

### Using Nested Selectors

```
.p {
 color: #000;
 font-size: 16px;
 span {
 color: #FF0000;
 em {
 text-decoration: underline;
 }
 }
}
```

The following code example shows the CSS output after compilation of Sass code:

### CSS Output After Compilation

```
.p {
 color: #000;
 font-size: 16px;
}
.p span {
 color: #FF0000;
}
.p span em {
 text-decoration: underline;
}
```

## Mixins

Mixins are one of the most powerful and time-saving features of Sass. Mixins allow you to group CSS declarations and reuse them throughout your code. This feature can help you to reduce the amount of code and avoid duplication.

The following code example demonstrates how to use a mixin to avoid duplication of code:

### Using a Mixin

```
/* Define a mixin */
@ mixin normalized-text() {
 font-weight: 300;
 line-height: 1.2;
}

/* Use a Mixin in multiple places */
.description {
 @include normalized-text();
 color: red;
}
.comment {
 @include normalized-text();
 color: grey;
}
```

The following code example shows the CSS output after compilation of Sass code:

### CSS Output After Compilation

```
.description {
 font-weight: 300;
 line-height: 1.2;
 color: red;
}

.comment {
 font-weight: 300;
 line-height: 1.2;
 color: grey;
}
```

Mixins in Sass act as functions - they allow you to pass in parameters and return sets of CSS properties. This allows mixins to be more flexible.

The following code example demonstrates how to pass a parameter to a mixin:

### Pass a Parameter to a Mixin

```
/* Define a mixin */
@mixin normalized-text($color) {
 font-weight: 300;
 line-height: 1.2;
 color: $color;
}
/* Use a Mixin in multiple places with different parameters */
.description {
 @include normalized-text(red);
}
.comment {
 @include normalized-text(grey);
}
```

The following code example shows the CSS output after compilation of Sass code:

### CSS Output After Compilation

```
.description {
 font-weight: 300;
 line-height: 1.2;
 color: red;
}

.comment {
 font-weight: 300;
 line-height: 1.2;
 color: grey;
}
```

## Control Directives

Control directives allow you to include certain styles only under specific conditions. They can be used inside mixins to include the same style with variations. Some of the control directives available in Sass are: @if, @else, @else if, @each and @for.

The following example demonstrates how to use control directives inside a mixin:

### Using Control Directives

```
@mixin season($season) {
 @if($season == summer) {
 color: red;
 }
 @else if ($season == autumn) {
 color: orange;
 }
 @else if ($season == winter) {
 color: grey;
 }
 @else {
 color: green;
 }
}

.description {
 @include season(summer);
}
```

The following code example shows the CSS output after compilation of Sass code:

### CSS Output After Compilation

```
.description {
 color: red;
}
```

## Functions

Sass exposes a wide range of functions that allow you to perform manipulations on styles such as increase or decrease the lightness of a given color, perform mathematic calculations such as **round** and **ceil** and more.

Following are examples of some functions available in Sass:

- **lighten**. Increases the lightness of a color by a certain percentage. For example, **lighten(#fe6700,20%)** will increase the lightness of a certain orange hue by 20% percent.
- **darken**. Increases the darkness of a color by a certain percentage. For example, **darken(#fe6700,20%)** will increase the darkness of a certain orange hue by 20% percent.

 **Additional Reading:** Full documentation of the functions available in Sass can be found at: <https://aka.ms/moc-20486d-m9-pg11>

The following code example demonstrates how to use the functions **lighten** and **darken**:

### Using Functions

```
$main-color: #80E619;

.main-headline {
 color: darken($main-color, 20%);
}

.sub-headline {
 color: lighten($main-color, 20%);
}
```

The following code example shows the CSS output after compilation of the Sass code:

### CSS Output After Compilation

```
.main-headline {
 color: #4d8a0f;
}

.sub-headline {
 color: #b3f075;
}
```

## Styling Applications with Less

Less is another CSS preprocessor that shares many features with Sass while providing some of its own. Like Sass it enhances CSS with variables, mixins, functions and more. While many functionalities are shared, the syntax is usually different.

### Using Less in your application

You can add a Less file to your application by adding a file ending with **.less** extension to your project. You will then need to compile the Less file to a CSS file.

- Less is a language that is compiled to CSS

- Installing Less by using npm:

```
npm install -g less
```

- Compiling a Less file:

```
lessc main.less main.css
```

 **Best Practice:** Place Less files inside of a dedicated Styles folder which is not located inside the **wwwroot** folder. Place the compiled CSS files inside the **wwwroot** folder.

To compile a Less file, you can use several techniques, which include:

- Install Less globally by using npm and then using the **lessc** command. This technique will be covered in this topic.
- Configure a task runner. This technique will be covered in Lesson 2, "Using Task Runners".

To install Less globally, you can run the following command:

### Installing Less

```
npm install -g less
```

The following code example demonstrates how to compile the **main.less** file to the **main.css** file:

### Compiling a Less File

```
lessc main.less main.css
```

## Variables

Variables allow you to store information that you want to repeat throughout your code. Less variables behave like Sass variables with a slight change of syntax. To define a new variable in Less you will use the @ sign.

The following code example demonstrates how to define and use Less variables:

### Using Less Variables

```
@highlights : #00FFFF;

.message {
 color: @highlights;
 font-size: 12px;
}

h1 {
 color: @highlights;
}
```

The following code example shows the CSS output after compilation of the Less code:

### CSS Output After Compilation

```
.message {
 color: #00FFFF;
 font-size: 12px;
}
h1 {
 color: #00FFFF;
}
```

## Nesting Styles

Nesting is a feature of Less that allows defining styles by using nested code. The syntax of Less nested selectors is the same as in Sass.

This following code example demonstrates how to define the style of elements by using nested selectors:

### Using Nested Selectors

```
.p {
 color: #000000;
 font-size: 16px;
 span {
 color: #FF0000;
 em {
 text-decoration: underline;
 }
 }
}
```

The following code example shows the CSS output after compilation of the Less code:

### CSS Output After Compilation

```
.p {
 color: #000000;
 font-size: 16px;
}
.p span {
 color: #FF0000;
}
.p span em {
 text-decoration: underline;
}
```

## Mixins

Mixins in Less behave and are declared differently than in Sass. Less mixins allow you to mix in values from one class into another class. This is a very important feature that can reduce the amount of code you have to write and to avoid duplication.

The following code example demonstrates the repetition of code inside CSS files that Less solves with mixins:

### CSS Without Mixins

```
.description {
 font-weight: 300;
 line-height: 1.2;
 color: red;
}

.comment {
```

```
font-weight: 300;
line-height: 1.2;
padding: 20px;
color: grey;
}
```

The following code example demonstrates how to use a mixin to avoid duplication of code:

### Using a Mixin

```
/* Define a class */
.normalized-text {
 font-weight: 300;
 line-height: 1.2;
}

/* Use a mixin in multiple places */
.description {
 .normalized-text();
 color: red;
}
.comment {
 .normalized-text();
 padding: 20px;
 color: grey;
}
```

### Functions

Less exposes a wide range of functions that allow you to perform manipulations on styles such as increase or decrease the lightness of a given color, get the dimension of an image in a specific URL, convert one unit to another, perform mathematic calculations such as round and ceil and more. Some methods might be like the ones that exist in Sass.

Following are examples of some functions that available in Less:

- **lighten**. Increases the lightness of a color by a certain percentage. For example, **lighten(#fe6700,20%)** will increase the lightness of a certain orange hue by 20% percent.
- **darken**. Increases the darkness of a color by a certain percentage. For example, **darken(#fe6700,20%)** will increase the darkness of a certain orange hue by 20% percent.
- **image-size**. Retrieves the width and height of a certain image. For example, if the values of the width and the height of the image.png file are 150px, calling **image-size("image.png")** will return 150px 150px.
- **convert**. Converts from one unit to another. For example, **convert(9s, "ms")** will convert 9s to 9000ms.

 **Additional Reading:** Full documentation of the functions available in Less can be found at: <https://aka.ms/moc-20486d-m9-pg1>

The following code example demonstrates how to use the **lighten** and **darken** functions:

### Using Less Functions

```
@main-color: #80E619;

.main-headline{
 color: darken(@main-color, 20%);
}
.sub-headline{
 color: lighten(@main-color, 20%);
}
```

The following code example shows the CSS output after compilation of the Less code:

### CSS Output After Compilation

```
.main-headline {
 color: #4d8a0f;
}
.sub-headline {
 color: #b3f075;
}
```

## Lesson 2

# Using Task Runners

In a modern web environment client-side code isn't always designed as simple JavaScript and CSS files. These days various technologies are being used which create additional steps in the development of client-side logic. Whether it can be the use of unconventional file types such as Sass or Less which require compilation, but in turn reduce the work for the developer, or it can be via processes such as bundling and minification that improve performance.

As web technologies grow, so does the need to support the myriad possibilities. Task runners were created for these purposes. Task runners are libraries and applications which can be used to define tasks and carry them out by using various plugins. This enables new technologies to add a plugin and become instantly viable. In addition, task runners can handle other development side tasks such as unit testing.

In this lesson, you will learn how to set up a task runner by using either gulp or Grunt, how to use them in compiling Sass, and how to connect the compilation to running during a Visual Studio build. You will later learn how to use gulp to perform bundling and minification and how to set up a watcher task to watch files as you work on them, enabling client-side development without a need to rebuild the solution.

It is important to understand that by properly utilizing task runners and Visual Studio integration, it is possible to focus more effort on developing client-side code and reducing the requirement to handle various menial tasks such as running scripts and performing operations by hand.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and state various use cases for task runners.
- Create and import tasks and run them in Grunt.
- Create tasks by using plugins and run them in gulp.
- Use gulp to fully import your client-side code and CSS folders into the **wwwroot** folder while utilizing advanced client-side technologies like Sass and Bootstrap.
- Use gulp to optimize your client-side code.

### Using Grunt

While developing a web application, there are operations that you may do many times.

Examples of such operations are minification of code and compilation of files. Instead of doing these operations yourself, you can use a task runner which will perform these operations for you.

In this lesson, two task runners will be introduced. In this topic, Grunt will be introduced. Another task runner, gulp, will be introduced in the next topic.

- Grunt is a task runner
- Used to perform operations such as bundling, minification and compilation
- Sections of Grunfile.js:
  - Project and task configuration information
  - Loading the grunt plugins and tasks
  - Custom tasks that are created by the user

Grunt operates by having each task run separately of every other task. Grunt itself only supports the basic infrastructure for running tasks and you will need to add additional plugins or configure the logic for the tasks in order to support them. As a result, Grunt supports a large number of additional task libraries each of which can be run as required, while avoiding having to deal with a bloated environment full of unnecessary code, and you can run a very bare environment by using only the plugins that you need.

Being a client-side task runner, Grunt will mainly be used to perform operations on client-side code. This can include things like performing minification (compressing JavaScript or CSS code to reduce space requirements, and therefore improve loading times), bundling (merging several different JavaScript or CSS files into one file, in order to reduce the number of requests), compiling nonstandard file types (such as Sass or TypeScript) and more.

In this topic, you will be introduced to Grunt and learn some basic functionality for it. You will learn how to compile Sass files by using Grunt, and how to integrate them to automatically run as you work on the project, enabling you to spend more time working on code, and less time managing tasks.

## Configuring the Environment

The first step required to start working with Grunt is to configure the environment with the necessities to hold a Grunt environment. To do this, you will first need to add Grunt as part of the npm configuration. This can be done inside the **package.json** file by adding Grunt to the **devDependencies** section.

The following code is an example of adding a development dependency to **package.json**:

### Grunt in package.json

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "devDependencies": {
 "grunt": "1.0.2"
 }
}
```

 **Note:** It is important to note that the version of Grunt in this example is the latest stable version at the time of writing this course. There might be a newer version at this time.

After adding Grunt to the **package.json** file, you need to add a new js file in the base folder of the project. This file should be named **Gruntfile.js**. Inside the **Gruntfile.js** file you need to create a wrapper function in a specific format, every single **Gruntfile.js** requires it, and cannot be used without it.

The following code is an example of a Gruntfile wrapper function:

### Gruntfile.js Wrapper

Using **gulp**

## Sections of Gruntfile.js

The wrapper inside the **Gruntfile.js** file is the external shell in which all of the Grunt code will run. There can be several different sections inside the wrapper function, which include:

- **Project and task configuration information.** The project and task configuration are handled inside the **grunt.initConfig** method. In here it is possible to configure various tasks and their behavior.
- **Loading the Grunt plugins and tasks.** Loading the Grunt plugins is done by using the **grunt.loadNpmTasks** method. This method accepts the name of a plugin which is then loaded. Once the plugin is loaded, it is possible to run the task.

- **Custom tasks which are created by the user.** This can range from a task designed to run several other tasks all the way to creating custom tasks with logic that is relevant for the specific application.

An example of a Grunt plugin is **grunt-sass**. This plugin allows compiling Sass files into CSS files. You can then use those CSS files in an ASP.NET Core MVC Application.

To use the **grunt-sass** plugin, you first need to add a dependency to it inside the **devDependencies** section of the **package.json** file.

The following code is an example of adding a Grunt plugin dependency:

### Adding a Grunt Plugin to package.json

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "devDependencies": {
 "grunt": "1.0.2",
 "grunt-sass": "2.1.0"
 }
}
```

Once the plugin is added to and downloaded by npm, you need to configure it in the **grunt.initConfig** method. The **grunt.initConfig** method receives an **options** parameter which is used to configure tasks. For example, for **grunt-sass**, a task for compiling Sass, the **sass** property in the options parameter should be assigned to. To compile files from a folder, you will need to first update the **dist** property (**dist** is a shorthand for distribute) and underneath to set the **files** property which is used to determine which files are copied.

The **files** property can accept an array of objects which can be used to declare the logic for Sass compilation, so that you can potentially copy from multiple sources into multiple destinations, or into the same destination.

Each object in the array is defined in the following format which is a standard Grunt format which can be used across many Grunt plugins:

- **expand**. When set to true, signifies working with directories rather than files.
- **cwd**. Short for current working directory, denotes the folder from which to run the script.
- **src**. Short for source, an array of regular expressions to match the files. Usually for Sass files ['\*\*/\*.scss'] or ['\*\*/\*.sass'] will be used.
- **dest**. Short for destination, determines the folder to which Sass files will be compiled.
- **ext**. Short for extension, determines under what file extension complied files will be saved. Usually .css for Sass.

The following code demonstrates how to configure **grunt-sass** inside the **grunt.initConfig** method:

### Gruntfile.js with Sass Configuration

```
module.exports = function(grunt) {
 grunt.initConfig({
 sass: {
 dist: {
 files: [
 {
 expand: true,
 cwd: 'Styles',
 src: ['**/*.scss'],
 dest: 'wwwroot/styles',
 ext: '.css'
 }
]
 }
 }
 });
}
```

```
 }
 });
};
```

In this example you can see that compilation for Sass is being configured from a directory (expand: true).

The directory you will compile from is Styles (cwd: 'Styles').

You will compile all files with the extension .scss inside of the folder (src: ['\*\*/\*.scss']).

The compiled files will be deployed to the wwwroot/styles folder (dest: 'wwwroot/styles').

The files will be deployed with the file extension css (ext: '.css').

 **Note:** The format of files is standard across Grunt itself. However, the specific order of configurations may change between the different plugins. Make sure to check the plugin you use carefully in case you need to make any changes.

Finally, once you have configured the parameters for the task, you will need to load it by calling the **grunt.loadNpmTasks** method, and supply **grunt-sass** as a parameter to the loading function.

The following code demonstrates a complete Gruntfile.js for compiling Sass files:

### Sass Compilation Task Configuration

```
module.exports = function(grunt) {
 grunt.initConfig({
 sass: {
 dist: {
 files: [
 {
 expand: true,
 cwd: 'Styles',
 src: ['**/*.scss'],
 dest: 'wwwroot/styles',
 ext: '.css'
 }
]
 }
 });
 grunt.loadNpmTasks("grunt-sass");
};
```

Once you have completed this, you should be able to see the new task inside the Visual Studio Task Runner Explorer window and you can run the task by right-clicking it and clicking on **Run**.

To demonstrate running the task configured, assume the following code of a Sass file named **myStyle.scss**, which is located inside a folder named **Styles** in the base folder of the application:

### Sass File

```
$main-color: #FF8934;

.myClass {
 background-color: $main-color;
}

#myElement {
 color: $main-color;
}
```

After running the task, a new CSS file named **myStyle.css** will be created under the **wwwroot/styles** folder.

## Registering a Custom Task

In addition to using existing plugins in the creation of tasks, you can also create your own tasks. Common tasks you may wish to create could involve various behaviors you desire such as adding tasks for logging or combining several tasks into one, which is also known as an alias task.

In order to create a task you can use the **grunt.registerTask** method. The basic syntax for creating a task is **grunt.registerTask(\*task name\*, [\*task description\*], \*task function\*)** with the description being optional. This allows creating a task which when it is run calls for a function.

The following code is an example of registering a custom task in the Gruntfile.js file:

### Register a Custom Task

```
module.exports = function(grunt) {
 grunt.registerTask('myTask', 'Saying hello', function() {
 grunt.log.write('Hello world...').ok();
 });
};
```

In this example, you are registering a task named **myTask** that, whenever it is run, the string **Hello World...** will be displayed in the task output screen.

Occasionally, you may wish for multiple tasks to occur at a time. In order to implement this, you will once again use the **grunt.registerTask** method, however this time, the syntax you will use is **grunt.registerTask(\*alias task name\*, [\*task description\*], \*task list\*)**, with the description being optional, and is often omitted entirely. Tasks like this are known as alias tasks, and in Visual Studio Task Runner Explorer they appear as such. By running this task all tasks in the list are executed in order.

The following code is an example of an alias task:

### Alias Task

```
module.exports = function(grunt) {
 grunt.initConfig({
 sass: {
 dist: {
 files: [
 {
 expand: true,
 cwd: 'Styles',
 src: ['**/*.scss'],
 dest: 'wwwroot/styles',
 ext: '.css'
 }
]
 }
 }
 });

 grunt.loadNpmTasks("grunt-sass");

 grunt.registerTask('myTask', 'Saying hello', function() {
 grunt.log.write('Hello world...').ok();
 });

 grunt.registerTask('myMultipleTasks', ['myTask', 'sass']);
};
```

In this example, you will see **myTask** being executed, followed by the Sass compilation task.

### Binding Tasks to Visual Studio Events

A powerful feature of Grunt is the ability to bind tasks to run alongside Visual Studio events. This is useful since it allows running tasks before a build occurs, allowing you to update any compiled files inside the project. Thus, you can run the application in a convenient fashion without needing to manually run tasks each time you update.

Alternatively, if you have tasks which you wish to run when the project is open, you can bind them to the project being opened and have the tasks run whenever you start Visual Studio.

To bind a task, you can right-click it in the Task Runner Explorer and select a binding to use. You can later change the order, or delete bindings, by right-clicking a task in the binding sub-window of the Task Runner Explorer.

## Using gulp

Another task runner that you can use is gulp. Unlike Grunt, a gulp task can perform several sequential operations on a single pipeline, leading to a very different development approach from Grunt.

Similarly to Grunt, gulp also supports only a very basic infrastructure for running tasks and additional plugins will need to be loaded in order to perform most operations. Configuration for gulp files is handled inside the tasks themselves, and since tasks run on a single pipeline, configuration often ends up being relatively simple. gulp supports a large number of plugins, which can grant gulp various different functionalities which can then be used inside of the tasks. This enables running a lean task environment where only required plugins are added.

gulp is intended to be used for client-side tasks, and is often used for bundling, minification, and compilation for various client-side technologies.

In this topic, you will learn the basics behind gulp, how to use it to compile Sass files, as well as how to integrate Sass compilation into the Visual Studio environment, preventing you from needing to compile on a regular basis.

- gulp is a task runner
- A gulp task can perform several sequential operations on a single pipeline
- Task configuration will be done inside a gulpfile.js file

### Configuring the Environment

As the first step in the process, you need to add gulp inside the **devDependencies** section of the npm configuration file, **package.json**. This will load up the JavaScript files required by gulp and will allow working with it.

The following code is an example of gulp configuration in package.json:

### gulp Dependency

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "devDependencies": {
 "gulp": "3.9.1"
 }
}
```

 **Note:** It is important to note that the version of gulp in this example is the latest stable version at the time of writing this course. There might be a newer version at this time.

After the dependency has been loaded, a **gulpfile.js** file should be created. Task configuration will be done inside the **gulpfile.js** file.

Inside **gulpfile.js** you should load up gulp by calling the **require('gulp')** function. The **require** function enables you to request that the **gulp** dependency will be loaded in the context of the current file. Doing this will enable you to start writing tasks.

The following code is an example of calling the **require** function in the gulpfile.js file:

### gulpfile.js

```
var gulp = require('gulp');
```

### Writing a Task

After setting up the environment it is possible to start creating gulp tasks. Unlike Grunt which tends to use tasks that are created in advance, and then configure them, in gulp, the configuration will usually be done as part of a created task. Therefore, every task that is run in gulp will always act as a custom task.

To add additional functionalities, you will need to load additional plugins. The first step is to add plugins to the **devDependencies** section of the **package.json** file. This will set npm to install them into the application. Afterwards, you will need to load them from inside the **gulpfile.js** file. This can be done by using the **require** method and providing it with the name of the plugin which you need to load.

An example of a gulp plugin is **gulp-sass**. This plugin allows compiling Sass files into CSS files. You can then use those CSS files in an ASP.NET Core MVC application.

The following code demonstrates adding the **gulp-sass** plugin to the **package.json** file:

### Adding a gulp plugin to package.json

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "devDependencies": {
 "gulp": "3.9.1",
 "gulp-sass": "4.0.1"
 }
}
```

The following code demonstrates adding the **gulp-sass** plugin to the **gulpfile.js** file:

### gulp-sass Plugin in gulpfile.js

```
var gulp = require('gulp');
var sass = require('gulp-sass');
```

After the **gulp-sass** plugin has been loaded, it will be available to use inside tasks.

By convention, all file configurations should be defined at the top of the **gulpfile.js** file, directly under the section for loading all plugins by using the **require** method. This is done so that you will be able to easily change them when the requirement changes. It is particularly helpful when using shared configurations across several tasks. However, this segment is entirely optional. Examples of relevant configurations could include file paths, task names, important strings and numbers and more. There isn't any one correct structure for configurations so you should create a JavaScript object that works for your requirements.

The following code demonstrates a **gulpfile.js** file with configurations:

### gulpfile.js Configuration Example

```
var gulp = require('gulp');
var sass = require('gulp-sass');

var paths = {
 webroot: "./wwwroot/"
};
paths.sass = "./Styles/*.scss";
paths.compiledCss = paths.webroot + "styles/";
```

In the example above, you can see that a shared path configuration is created, which sets the **webroot** property to hold the path to the **wwwroot** folder. Additionally, it is then used to create a path to where the Sass files are stored and the destination for compiled Sass files.

After the setup is completed, a task can be created by using the **gulp.task** function. This function uses the following syntax: **gulp.task(\*task name\*, [\*dependency tasks\*], \*task function\*)**. The task name being the name of the task which will be created, dependency tasks is a list of other tasks which will be run as part of this task, and the task function is the code which will be run in the task. Both the dependency tasks and the task function are optional parameters. You can choose to use one or the other or both as needed.

Inside the **gulp.task** function you can add your code and use it to perform any tasks you wish. Commonly gulp functions will perform various operations by creating a memory stream, writing into it, and finally closing it turning it into a read-only stream. By using the **pipe** method on the stream, it will write the current stream into the stream created by the following function, allowing it to continue working from the point at which the last function finished. The **pipe** function receives a method as a parameter which will receive the stream from the previous method in the pipeline.

You will often begin gulp tasks by invoking the **gulp.src** method. The first parameter of this method is a regex file path or potentially a collection of regex file paths. This allows loading one or even more files as needed to perform your desired operation. You can then use the **pipe** function to continue processing the files.

At the end of the process, the **gulp.dest** method will frequently be called to save the stream that was previously created into files. It receives a folder into which it will write all files currently stored inside the memory stream. It is usually the last method in the pipeline, however, you can still use the **pipe** function to continue using the stream.

When loading the **gulp-sass** plugin, it is also possible to use the **sass** method. By using a stream that it receives from the pipe, the **sass** method compiles Sass files into CSS files and returns a stream which can be used with the **pipe** method to continue the process. It is also possible to attach to the **on** event.

The following code demonstrates a complete gulp task:

### gulp Sass Compilation

```
var gulp = require('gulp');
var sass = require('gulp-sass');

var paths = {
 webroot: "./wwwroot/"
};

paths.sass = "./Styles/*.scss";
paths.compiledCss = paths.webroot + "styles/";

gulp.task("sass", function() {
 return gulp.src(paths.sass)
 .pipe(sass().on('error', sass.logError))
 .pipe(gulp.dest(paths.compiledCss));
});
```

In this example, a task is created that reads all of the Sass files under the **styles** folder of the application. It then compiles the Sass files into CSS files which are then placed in the **styles** folder inside of the **wwwroot** folder. At this point, you can use the Visual Studio Task Runner Explorer to run a newly created task.

### Running Multiple Tasks

By providing the gulp **task** function with a list of task names you can run several tasks at once. When running multiple tasks in gulp, each task will be run in parallel to the other. This means that tasks which need to run in a specific order will need to be handled specifically to run in that order. It is also possible to optionally run an additional code by adding a function after the task. This code will only execute once all other tasks have been completed. This can allow setting up chains of tasks running in order.

The following code demonstrates a task running multiple tasks in gulp:

### Multiple Tasks in gulp

```
var gulp = require('gulp');

gulp.task("one", function() {
 console.log("one");
});

gulp.task("two", function() {
 console.log("two");
});

gulp.task("three", function() {
 console.log("three");
});

gulp.task("all", ["one", "two", "three"], function() {
 console.log("four");
});
```

In this example, calling the **all** task will run tasks **one**, **two** and **three**, before finally printing the string **four** in the Task Runner Console.

### Binding Tasks to Visual Studio Events

As with Grunt, it is possible to bind gulp tasks to Visual Studio events. This can be done in the same way, by right-clicking a task name in the Task Runner Explorer window and selecting an event to which you can bind.

## Demonstration: How to Use gulp to Compile Sass to CSS

In this demonstration, you will learn how to set up a basic task runner by using gulp to compile a Sass file to a CSS file.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use gulp to Compile Sass to CSS" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD09\\_DEMO.md#demonstration-how-to-use-gulp-to-compile-sass-to-css](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD09_DEMO.md#demonstration-how-to-use-gulp-to-compile-sass-to-css).

## Bundling and Minification

One of the most frequent usages of task runners is to perform bundling and minification for the client side. In this topic, you will learn how you can perform bundling and minification by using gulp and the Visual Studio Task Runner Explorer.

A big advantage to bundling and minification is that by reducing the number of files (bundling), it is possible to ensure that less file requests are made to the server and by reducing the size of files (minification), requests to the server occur faster. Both of these help web applications become dynamic and responsive, reducing the potential of a user becoming frustrated by the application for being "slow".

To perform bundling and minification, the following gulp plugins can be added:

- **gulp-concat**. Used for combining multiple files performing a bundling operation.
- **gulp-cssmin**. Minifies CSS files.
- **gulp-uglify**. Minifies JavaScript files.

To add these gulp plugins to a web application, you can add them to the **devDependencies** section of the npm configuration file, **package.json**.

The following code demonstrates how to add gulp plugins to the **package.json** file:

### Adding gulp Plugins to package.json

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "dependencies": {
 "bootstrap": "4.1.3",
 "jquery": "3.3.1",
 "popper.js": "1.14.3"
 },
 "devDependencies": {
 "gulp": "3.9.1",
 "gulp-sass": "4.0.1",
 "gulp-concat": "2.6.1",
 "gulp-cssmin": "0.2.0",
 "gulp-uglify": "3.0.1"
 }
}
```

• One of the most frequent usages of task runners is to perform bundling and minification

- **Bundling**. Reduces the number of files
- **Minification**. Reduces the size of files

• To perform bundling and minification, the following gulp plugins can be added:

- **gulp-concat**. Used for combining multiple files performing a bundling operation
- **gulp-cssmin**. Minifies the CSS files
- **gulp-uglify**. Minifies the JavaScript files

```
}
```

By using these plugins, you will be able to perform bundling and minification. Throughout this topic, you will see several separate minification and bundling tasks and you will learn additional possibilities for working with gulp.

The following code example demonstrates how to minify a CSS file which is located in the **node\_modules** folder and add the result to the **wwwroot** folder:

### An Example of CSS File Minification

```
var gulp = require('gulp');
var concat = require("gulp-concat");
var cssmin = require("gulp-cssmin");

var paths = {
 webroot: "./wwwroot/",
 nodeModules: "./node_modules/"
};

paths.bootstrapCss = paths.nodeModules + "bootstrap/dist/css/bootstrap.css";
paths.vendorCssFileName = "vendor.min.css";
paths.destinationCssFolder = paths.webroot + "styles/";

gulp.task("minify-vendor-css", function() {
 return gulp.src(paths.bootstrapCss)
 .pipe(concat(paths.vendorCssFileName))
 .pipe(cssmin())
 .pipe(gulp.dest(paths.destinationCssFolder));
});
```

In this example, the **bootstrap.css** file is read from the **node\_modules** folder and saved as **vendor.min.css** by calling the **concat** function. Then the CSS content inside it is minified by calling the **cssmin** function. Finally, the file is written into the **styles** folder of the **wwwroot** folder. At this point, the file can easily be linked to from a layout or a view. A major advantage of this approach is that if new CSS libraries are added, they can easily be added to the **gulpfile.js** without affecting the client.

The following code is an example of compiling, bundling and minifying Sass files:

### Sass Preprocessing

```
var gulp = require('gulp');
var concat = require("gulp-concat");
var cssmin = require("gulp-cssmin");
var sass = require('gulp-sass');

var paths = {
 webroot: "./wwwroot/",
 nodeModules: "./node_modules/"
};

paths.sassFiles = "./Styles/*.scss";
paths.compiledCssFileName = "style.min.css";
paths.destinationCssFolder = paths.webroot + "styles/";

gulp.task("minify-sass", function() {
 return gulp.src(paths.sassFiles)
 .pipe(sass().on('error', sass.logError))
 .pipe(concat(paths.compiledCssFileName))
 .pipe(cssmin())
 .pipe(gulp.dest(paths.destinationCssFolder));
});
```

In this code example, all **.scss** files inside of the application **styles** folder are read. Next, all of the **.scss** files are compiled into regular CSS files. Then, the CSS files are combined to a single file named **style.min.css**. Afterwards, the combined CSS file is minified. Finally, the file is written into the **styles** folder of the **wwwroot** folder. At this point, you can link this file to the layout or a view and it will be fully usable.

The following code is an example of bundling and minifying multiple client-side JavaScript files:

### JavaScript Bundling and Minification

```
var gulp = require('gulp');
var concat = require("gulp-concat");
var uglify = require("gulp-uglify");

var paths = {
 webroot: "./wwwroot/",
 nodeModules: "./node_modules/"
};

paths.bootstrapjs = paths.nodeModules + "bootstrap/dist/js/bootstrap.js";
paths.jqueryjs = paths.nodeModules + "jquery/dist/jquery.js";
paths.vendorJsFiles = [paths.bootstrapjs, paths.jqueryjs];
paths.vendorJsFileName = "vendor.min.js";
paths.destinationJsFolder = paths.webroot + "scripts/";

gulp.task("minify-vendor-js", function() {
 return gulp.src(paths.vendorJsFiles)
 .pipe(concat(paths.vendorJsFileName))
 .pipe(uglify())
 .pipe(gulp.dest(paths.destinationJsFolder));
});
```

This code example demonstrates how to serve both Bootstrap and jQuery in a single minified JavaScript file. This task begins by reading both **bootstrap.js** and **jquery.js** by using an array of file paths. The task then merges both files under the file name **vendor.min.js**, and then it minifies the resulting file by calling the **uglify** function. Finally, the file is written into **scripts/vendor.min.js**. In the future, should the need arise to add more JavaScript files, they can be added into the **gulpfile.js** without needing to update the layout or a view.

 **Note:** It is crucial to note that the logic for minifying CSS and JavaScript files is different, necessitating separate minification plugins. However, since **gulp-concat** only puts files together and renames them it is possibly to use it to bundle both.

The following code is another example of bundling and minifying JavaScript files:

### Bundling and Minifying JavaScript Files

```
var gulp = require('gulp');
var concat = require("gulp-concat");
var uglify = require("gulp-uglify");

var paths = {
 webroot: "./wwwroot/",
 nodeModules: "./node_modules/"
};

paths.jsFiles = "./Scripts/*.js";
paths.minifiedJsFileName = "scripts.min.js";
paths.destinationJsFolder = paths.webroot + "scripts/";

gulp.task("minify-js", function() {
```

```

 return gulp.src(paths.jsFiles)
 .pipe(concat(paths.minifiedJsFileName))
 .pipe(uglify())
 .pipe(gulp.dest(paths.destinationJsFolder));
 });
}

```

This code example demonstrates a task that reads all files in the **Scripts** directory of the application, combines them into a file called **scripts.min.js**, minifies it and finally saves it in **wwwroot** inside the **scripts** folder, which is easily linked to from the layout or a view.

 **Best Practice:** It is considered a best practice to keep code from **node\_modules** separate from the application code in the process of bundling and minification. This helps to troubleshoot problematic areas within the application code. However, it is entirely possible to combine **node\_modules** code with the application code.

## Integrating with Build

It is a good idea for JavaScript and CSS files to be bundled and minified whenever a new build is running. This will enable using the latest versions of JavaScript and CSS files, providing a hassle-free environment and allowing complete focus on the coding aspect, while not having to deal with manually running tasks. To achieve this, you can create a singular task assigned to running all the separate subtasks.

The following code demonstrates tasks running multiple subtasks:

### Multiple Tasks

```

gulp.task("all-css", ["minify-vendor-css", "minify-sass"]);

gulp.task("all-js", ["minify-vendor-js", "minify-js"]);

gulp.task("minify-all", ["all-js", "all-css"], function () {
 console.log("All tasks completed!");
});

```

In this example, you can see that three new tasks have been added. The first performs both CSS bundling and minification tasks and the second does the same for JavaScript. Meanwhile, the third task calls both and when all subtasks are done, it prints: **All Tasks Completed** to the task runner output.

By binding the final task inside the task runner to build start, you can ensure that whenever the application is run, the latest version of files will be served and available.

### Watcher Tasks

An additional tool at your disposal is watcher tasks. Watcher tasks are tasks that are usually run when the project is loaded and continue running until it is closed. The purpose of watcher tasks is to watch for any changes occurring in files and if a change occurs, to run the appropriate task. For example, this can allow making real-time changes in the .scss and .js files and have the appropriate compilation, bundling, and minification occurs immediately. This can help reduce load during the build process and to allow testing client-side file changes without recompiling server-side code.

A watcher is configured by using the **gulp.watch()** method. The first parameter that will be supplied to the watch is a file or directory regex or a list of such. The second parameter is an array of tasks which will be run by the watcher.

Watchers are commonly bound to the **project.open** event of the Task Runner Explorer, allowing them to run in the background and perform their operations as the developer engages in their day to day work.

The following code is an example of watchers:

#### gulp Watcher Tasks

```
gulp.task("sass-watcher", function() {
 gulp.watch(paths.sassFiles, ["minify-sass"]);
});

gulp.task("js-watcher", function() {
 gulp.watch(paths.jsFiles, ["minify-js"]);
});
```

In this example, you can see that a watcher has been applied to the **styles** folder of the application, and another watcher to the **scripts** folder. When Sass code inside the **styles** folder is added, removed or changed, the **minify-sass** task will be run. Meanwhile, whenever a JavaScript file inside the application's **scripts** folder is added, removed or changed, the **minify-js** task will be run.

## Lesson 3

# Responsive Design

Responsive web design is used to make sure your application looks good on any screen and device. It is an approach that says that the design should respond to users' behavior based on screen size, platform, and orientation. In this lesson, you are going to learn different techniques to adapt your website to various screen sizes and devices.

### Lesson Objectives

After completing this lesson, you will be able to:

- Ensure that a web application displays correctly on different devices.
- Use media queries to apply different CSS for different screen sizes.
- Use the Bootstrap grid system to layout the content of the pages.
- Use CSS flexbox to create flexible web applications.

### The HTML5 Viewport Attribute

It is possible to customize your web application's display based on the capabilities and specifications of a web browser or a device by using adaptive rendering.

Mobile browsers such as Microsoft Edge use the **viewport** attribute to render webpages in a virtual window. This virtual window is usually wider than the application screen. The **viewport** attribute helps eliminate the need to reduce the size of the layout for each page. Reducing the size of the layout can distort the display of non-mobile-optimized web applications. Creating the application interface by using the **viewport** attribute enables users to zoom into the different areas of a webpage.

The **viewport** tag is a meta tag that helps you to control the width and height of webpages.

The following example illustrates how to use the **viewport** tag:

#### Using the **viewport** Tag in a View

```
<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
 <title>Index</title>
</head>
<body>
</body>
</html>
```

The **viewport** attribute helps render webpages in a virtual window in mobile devices

```
<meta name="viewport" content="width=device-width,
initial-scale=1, maximum-scale=1">
```



The **width** and **height** properties help you to specify the width and height of the virtual viewport window in pixels. You can use the **device-width** keyword to enable the content to fit the default screen size of the browser.

The **initial-scale** property controls the initial scale or zoom level of a webpage. You can use other properties such as **maximum-scale**, **minimum-scale**, and **user-scalable** to control how the user can zoom in or out of the page.



**Additional Reading:** For more information about the **viewport** attribute, refer to:  
<https://aka.ms/moc-20486d-m9-pg2>

## CSS Media Queries

To support different browsers and devices, you may sometimes need to apply different CSS styles in your application. HTML5 includes CSS media queries, which are special selectors that begin with **@media**. Media queries allow conditional application of CSS styles based on the device conditions or browser capabilities. You can apply media queries in CSS and HTML.

The following example illustrates a media query in CSS:

### Using a Media Query

```
@media only screen and (max-width: 500px) {
 header{
 float: none;
 }
}
```

Characteristics of media queries:

- Media queries are special selectors that begin with **@media**
- You can also apply media queries in the **<link>** elements
- Media queries support properties that allow you to specify the size details of the targeted display area

```
@media only screen and (max-width: 500px) {
 header{
 float: none;
 }
}
```

You can also apply a media query in the **<link>** element. The following example illustrates how to include a media query in a **<link>** element:

### Using a Media Query in the link Element

```
<link rel="stylesheet" type="text/css" href="smallscreen.css" media="only screen and
(max-width: 500px)" />
```

You can use CSS media queries to apply CSS styles when the screen size is less than 500 pixels. However, you can use CSS media queries only for the screen layout, but not the print layout.

The following table describes properties that you can include in a media query.

Property	Description
<b>width</b>	The width of the targeted display area, which includes the browser window in desktop and mobile devices. In desktop computers, when you resize the browser window, the width of the browser changes. However, on most mobile browsers, you cannot resize the browser window. This implies that the width of the browser remains constant.
<b>height</b>	The height of the targeted display area, which includes the browser window in desktop and mobile devices.
<b>device-width</b>	The width of the entire screen of a device. For a desktop with a screen resolution of 1,024x768, the device-width is usually 1,024 pixels.

<b>device-height</b>	The height of the entire screen of a device. For a desktop with a screen resolution of 1,024x768, the device-height is usually 768 pixels.
<b>orientation</b>	The orientation of the device. If the device-width is larger than <b>the device-height</b> , the orientation value is set to <b>landscape</b> ; otherwise, it is set to <b>portrait</b> .
<b>aspect-ratio</b>	The ratio of the <b>width</b> and <b>height</b> properties.
<b>device-aspect-ratio</b>	The ratio of the <b>device-width</b> and <b>device-height</b> properties. The following example illustrates the device-aspect-ratio for a device with a screen resolution of 1,280x720. <pre>@media screen and (device-aspect-ratio: 16/9) { } @media screen and (device-aspect-ratio: 1280/720) { } @media screen and (device-aspect-ratio: 2560/1440) { }</pre>
<b>color</b>	The number of bits per color component of the device. If the device is not a color device, the value is zero.
<b>color-index</b>	The number of entries in the color lookup table, of the output device.
<b>monochrome</b>	The number of bits per pixel in a monochrome frame buffer. For non-monochrome devices, this value is zero.
<b>resolution</b>	The resolution of the output device or the density of the pixels. The common units for this property include dpi and dpcm.
<b>scan</b>	The scanning process of TV output devices.
<b>grid</b>	The property that detects whether the output is in the grid or bitmap format. Grid-based devices return a value of one; all other devices return a value of zero.

## The Bootstrap Grid System

When creating web applications, there is usually a need to layout the content of the pages in columns and rows. For example, a website can have a row for the header, a row for the body that contains the sidebar column and the content, and finally a row for the footer. These rows and columns can be responsive and compatible with multiple devices.

In the past, this was done with tables, but a better practice today is to use a grid system with CSS. Bootstrap includes a pre-built grid system that allows you to layout the content easily without writing any extra CSS code. Bootstrap allows you to divide the layout into up to 12 columns and display your content within them.

In the Bootstrap grid system, the most basic layout element is a container. Each Bootstrap grid system should have a container. A container is an element with one of the following classes:

- Bootstrap grid is a layout system which includes 12 columns
- It uses containers, rows and columns to organize and align the content

```
<div class="container">
 <div class="row">
 <div class="col">
 <h3>First Column</h3>
 </div>
 <div class="col">
 <h3>Second Column</h3>
 </div>
 </div>
</div>
```

- **container**. Use this class when you want to have a responsive container in which its **min-width** property is changed at predefined media query ranges called breakpoints. The following are the breakpoints which exist in Bootstrap:
  - **xs**. Used for extra small devices (portrait phones). min-width is 0px.
  - **sm**. Used for small devices (landscape phones). min-width is 576px.
  - **md**. Used for medium devices (tablets). min-width is 768px.
  - **lg**. Used for large devices (desktops). min-width is 992px.
  - **xl**. Used for extra-large devices (large desktops). min-width is 1200px.
- **container-fluid**. Use this class when you want the grid to be spread across the full width of the viewport.

In the container, you can create rows. A row is an element with the **row** class. Each row can include columns. A column is an element with the **col** class.

The following code example demonstrates how to create a container with rows and columns:

### Using Bootstrap Grid System in a View

```
@{
 ViewBag.Title = "Bootstrap Grid System Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap grid system</h3>

<div class="container">
 <div class="row">
 <div class="col">
 <h3>First Column</h3>
 </div>
 <div class="col">
 <h3>Second Column</h3>
 </div>
 <div class="col">
 <h3>Third Column</h3>
 </div>
 </div>
 <div class="row">
 <div class="col">
 Second Row - First column content.
 </div>
 <div class="col">
 Second Row - Second column content.
 </div>
 <div class="col">
 Second Row - Third column content.
 </div>
 </div>
</div>
```

In the example above, a grid with two rows is created. Each row contains three equal-width columns in all kind of devices (extra small, small, medium, large and extra-large).

It is possible to place different column sizes inside the same grid row. For example, the class **col-4** defines a column that takes four places and a **col-2** column defines a column that takes only two places.

The basic CSS classes that available for columns are: **col-1**, **col-2**, **col-3**, **col-4**, **col-5**, **col-6**, **col-7**, **col-8**, **col-9**, **col-10**, **col-11**, **col-12**. The number beside the col- defines how many grid columns the column will spread across.

The following code example demonstrates how to create a Bootstrap grid system having columns with different width:

### Non-equal Width Columns

```
@{
 ViewBag.Title = "Bootstrap Grid System Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap grid system</h3>

<div class="container">
 <div class="row">
 <div class="col-4">
 <h3>4 Columns width</h3>
 </div>
 <div class="col-6">
 <h3>6 Columns width</h3>
 </div>
 <div class="col-2">
 <h3>2 Columns width</h3>
 </div>
 </div>
 <div class="row">
 <div class="col-3">
 <h3>3 Columns width</h3>
 </div>
 <div class="col-6">
 <h3>6 Columns width</h3>
 </div>
 <div class="col-3">
 <h3>3 Columns width</h3>
 </div>
 </div>
</div>
```

In the example above, in the first row, the width of the first column is 33.33%, the width of the second column is 50%, and the width of the third column is 16.67%. In the second row, the width of the first column is 25%, the width of the second column is 50%, and the width of the third column is 25%.

It is possible to set the Bootstrap grid system to behave differently based on the device on which the application is running. For small devices, use the **col-sm** class, for medium devices, use the **col-md** class, for large devices, use the **col-lg** class, and for extra-large devices, use the **col-xl** class.

The following code example demonstrates how to create a responsive Bootstrap grid system:

### A Responsive Bootstrap Grid System

```
@{
 ViewBag.Title = "Bootstrap Grid System Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap grid system</h3>

<div class="container">
 <div class="row">
 <div class="col-12 col-md-4">
 <h3>first row, first column</h3>
 </div>
 <div class="col-6 col-md-8">
 <h3>first row, second column</h3>
 </div>
 </div>
 <div class="row">
 <div class="col-6 col-sm-4">
```

```
<h3>second row, first column</h3>
</div>
<div class="col-6 col-sm-4">
 <h3>second row, second column</h3>
</div>
<div class="col-6 col-sm-4">
 <h3>second row, third column</h3>
</div>
</div>
</div>
```

In the example above, in the first row for medium, large and extra-large devices the width of the first column is 33.33% and the width of the second column is 66.67%. In extra-small and small devices, the width of the first column is 100% and the width of the second column is 50%. In the second row, for extra-small devices, the width of each column is 50%, and for all other devices, the width of each column is 33.33%.

## Alignment

It is possible to vertically align the columns within the row in three main ways. To do so you need to add a class to the row element:

- **align-items-start**. Aligns the columns to the top.
- **align-items-center**. Aligns items to the center.
- **align-items-end**. Aligns items to the bottom.

Also, it is possible to align the items horizontally inside the row, using the following classes:

- **justify-content-start**. Align columns to the start of the row.
- **justify-content-center**. Align columns to the center of the row.
- **justify-content-end**. Align columns to the end of the row.
- **justify-content-around**. Spread the empty space around the columns.
- **justify-content-between**. Spread the empty space between the columns.

This example shows how to align columns vertically and horizontally inside rows:

### Grid Alignment

```
@{
 ViewBag.Title = "Bootstrap Grid System Example";
}
<h2>@ViewBag.Title</h2>
<h3>This page shows a Bootstrap grid system</h3>

<div class="container">
 <div class="row align-items-center justify-content-center">
 <div class="col">
 Column aligned to center
 </div>
 <div class="col">
 Column aligned to center
 </div>
 <div class="col">
 Column aligned to center
 </div>
 </div>
</div>
```



**Additional Reading:** Full documentation of Bootstrap grid system is located at:  
<https://aka.ms/moc-20486d-m9-pg12>

## Demonstration: How to Use the Bootstrap Grid System

In this demonstration, you will learn how to use the Bootstrap grid system in an ASP.NET Core web application.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Use the Bootstrap Grid System" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D\\_MOD09\\_DEMO.md#demonstration-how-to-use-the-bootstrap-grid-system](https://github.com/MicrosoftLearning/20486D-DevelopingASPARTMVCWebApplications/blob/master/Instructions/20486D_MOD09_DEMO.md#demonstration-how-to-use-the-bootstrap-grid-system).

## Applying the Flexbox Layout

CSS flexbox is a module added to the CSS and is supported by all modern browsers. It provides an efficient way to layout and align items inside a specific parent container. The Bootstrap grid system, which was introduced earlier in this lesson, is based on CSS flexbox.

### Parent Container

Parent container in context of flexbox module is an HTML element that has other HTML elements within it and has the property **display: flex** applied. One of the main ideas of flexbox is that the parent container can alter its children items width, height, and order to fill the available space within it in diverse ways.

Parent container properties are:

- **flex-direction: row | row-reverse | column | column-reverse;**

Defines the direction in which the container's children flow.

- **flex-wrap: nowrap | wrap | wrap-reverse;**

Flexbox items will always try to fit in one row or column. You can change this behavior and allow the items to wrap to second line or column when the available space ends by using flexbox-wrap.

- **justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly;**

Defines how the items are aligned on the main axis.

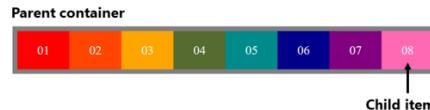
- **align-items: flex-start | flex-end | center | baseline | stretch;**

Defines how the items are laid on the cross axis on the baseline.

- **align-content: flex-start | flex-end | center | space-between | space-around | stretch;**

Similar to the justify-content but on the cross axis. Allows to align items inside a flex container when there is extra space on the cross-axis.

- All modern browsers support Flexbox
- It provides an efficient way to layout and align items inside a specific parent container
- Using **display: flex** a parent container can alter its children items width, height and order to fill the available space within it
- Each direct child inside the flex container can override the behavior specified by the parent container



The following code example demonstrates how to use parent container properties to lay out the content of a view:

### Using Flexbox Parent Container Properties

```
@{
 Layout = null;
}

<!DOCTYPE html>

<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Index</title>
 <style>
 .container {
 display: flex;
 justify-content: center;
 align-content: center;
 flex-wrap: wrap;
 }

 .item {
 width: 200px;
 background-color: grey;
 }
 </style>
</head>
<body>
 <div class="container">
 <div class="item">01</div>
 <div class="item">02</div>
 <div class="item">03</div>
 <div class="item">04</div>
 <div class="item">05</div>
 <div class="item">06</div>
 <div class="item">07</div>
 <div class="item">08</div>
 </div>
</body>
</html>
```

### Child Items

Each direct child inside the flex container can override the behavior specified by the container by using the properties described below:

- **order: -n...0...n;**  
Allows to target individual items and change where they appear in the visual order.
- **flex-grow: 0...n; default 0;**  
This defines the ability for a flex item to grow if necessary. It dictates what amount of the available space inside the flex container the item should take up.
- **flex-shrink: 0...n; default 0;**  
This defines the ability for a flex item to shrink if necessary. It determines how much the flex item will shrink relative to the rest of the flex items in the flex container.
- **flex-basis: length | auto;**  
This defines the default size of an element before the remaining space is distributed. *The **flex grow** and **flex shrink** are relative to flex basis.*

- *align-self: flex-start | flex-end | center | space-between | space-around | stretch;*

This allows the default alignment (specified by align-items) to be overridden for individual child item.

This example shows how to use child item properties to lay out the content of the page inside the container:

### Using Flexbox Child Item Properties

```
@{
 Layout = null;
}

<!DOCTYPE html>

<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Index</title>
 <style>
 .container {
 display: flex;
 justify-content: center;
 align-items: center;
 height: 800px;
 }

 .box {
 width: 100px;
 }

 .box-4 {
 order: -1;
 align-self: flex-start;
 }

 .box-2 {
 flex-grow: 2;
 }
 </style>
</head>
<body>
 <div class="container">
 <div class="box box-1">01</div>
 <div class="box box-2">02</div>
 <div class="box box-3">03</div>
 <div class="box box-4">04</div>
 <div class="box box-5">05</div>
 <div class="box box-6">06</div>
 <div class="box box-7">07</div>
 <div class="box box-8">08</div>
 </div>
</body>
</html>
```

# Lab: Client-Side Development

## Scenario

You have been asked to create a web-based ice cream application for your organization's customers. The application should have a page showing all kinds of ice creams in stock, and a purchase page which will allow customers to purchase ice cream. To style the application, you decided to use Bootstrap and Sass. You have decided to use gulp to compile, minify and bundle files.

## Objectives

After completing this lab, you will be able to:

- Install gulp by using npm.
- Write tasks by using gulp.
- Style the application by using Sass and Bootstrap.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD09\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD09_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD09\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD09_LAK.md).

## Exercise 1: Using gulp to Run Tasks

### Scenario

In this exercise, you will first install gulp using npm. You will then create a JavaScript file named **gulpfile.js**. After that, you will write tasks in the **gulpfile.js** file to bundle and minify JavaScript files. Finally, you will write a watcher task to track for any changes occurring in files which are located in the **Scripts** folder.

The main tasks for this exercise are as follows:

1. Use npm to install gulp.
2. Write a task to copy a JavaScript file.
3. Run the task.
4. Write a task to minify a JavaScript file.
5. Write a task to bundle and minify all JavaScript files in a folder.
6. Add a watcher task.
7. Run the tasks.

## Exercise 2: Styling by Using Sass

### Scenario

In this exercise, you will first create a Sass file named **main.scss** and fill its content. After that, you will create a gulp task to compile the Sass file to a CSS file. Then you will create a gulp watcher task so compilation of the Sass file to a CSS file will be done automatically when the Sass file is changed.

The main tasks for this exercise are as follows:

1. Add a new Sass file to the project.
2. Add gulp tasks to handle the Sass files.
3. Run a task.

## Exercise 3: Using Bootstrap

### Scenario

In this exercise, you will first update the **min-vendor:js** task that bundles and minifies JavaScript files to include the JavaScript files of Bootstrap. After that, you will add a task to handle the CSS files of Bootstrap. You will then run the tasks to create the **vendor.min.css** file and to update the **vendor.min.js** file. After that, you will style the layout by using Bootstrap. Finally, you will create a view to buy an ice cream and style it by using Bootstrap.

The main tasks for this exercise are as follows:

1. Update gulpfile.js to handle Bootstrap.
2. Run the tasks.
3. Style the application by using Bootstrap.
4. Run the application.

**Question:** A developer in your team added a function to the **payment-calc.js** file which is located in the **Scripts** folder. He was surprised to see the **script.min.js** file was automatically updated. Can you explain to him why the **script.min.js** file was automatically updated?

**Question:** A member of your team was told the application is styled by using Bootstrap. However, when she looked at the layout of the application, she didn't see any link to Bootstrap. Can you explain to her how the application is linked to Bootstrap?

# Module Review and Takeaways

In this module, you saw how to use several client-side tools and packages inside an ASP.NET Core web application. You first saw what is Bootstrap and then you saw several Bootstrap components that can be used in your application. You then learned how to style your application by using Sass and Less. After that, the Grunt and gulp task runners were covered, which help performing operations such as compiling Sass and Less files to CSS and bundling and minifying CSS and JavaScript files. Finally, different techniques to have a responsive web application were introduced, including CSS media queries, the Bootstrap grid system and flexbox.

## Review Questions

**Question:** What are the differences and similarities between Sass and Less?

**Question:** What tools and techniques would you use to adapt your web application for mobile and tablet devices?

## Best Practices

- Use Sass or Less to add features to CSS, such as variables, nested rules, functions, inheritance, importing styles and operators which improve the maintainability of large and complex applications.
- Use task runners not only to compress or compile files, but to add a variety of additional behaviors such as code quality tools and client-side unit testing.
- Use the **viewport** attribute to customize your web application's display based on specifications of the user's web browser.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The Bootstrap component does not look as it supposed to.	
Properties included in the flexbox module are not working.	

# Module 10

## Testing and Troubleshooting

### Contents:

Module Overview	10-1
Lesson 1: Testing MVC Applications	10-2
Lesson 2: Implementing an Exception Handling Strategy	10-14
Lesson 3: Logging MVC Applications	10-24
Lab: Testing and Troubleshooting	10-31
Module Review and Takeaways	10-34

## Module Overview

Software systems such as web applications are complex and require multiple components, which are often written by different developers, to work together. Incorrect assumptions, inaccurate understanding, coding errors, and many other sources can create bugs that result in exceptions or unexpected behavior. To improve the quality of your web application and create a satisfying user experience, you must identify bugs from any source and eliminate them.

Traditionally, testers perform most of the testing at the end of a development project. However, it has recently become widely accepted that testing throughout the project life cycle improves quality and ensures that there are no bugs in production software. You need to understand how to run tests on small components of your web application to ensure that they function as expected before you assemble them into a complete web application.

It is also important that you know how to handle exceptions while they occur. While an application is running, you may encounter unexpected occurrences. It is important that you manage your exceptions correctly and provide a good user feedback while avoiding leaking information about the application structure.

Finally, by using logs throughout the application, you can monitor user activities that might lead to unexpected issues and then you can find solutions to bugs, which you normally would be unsure how to reproduce, by following flows which occurred on the production environment and finding additional errors.

### Objectives

After completing this module, you will be able to:

- Run unit tests against the Model–View–Controller (MVC) components, such as model classes and controllers, and locate potential bugs.
- Build a Microsoft ASP.NET Core MVC application that handles exceptions smoothly and robustly.
- Run logging providers that benefit your applications and run them by using a common logging API.

## Lesson 1

# Testing MVC Applications

A unit test is a procedure that instantiates a component that you have written, calls one aspect of the functionalities of the component, and checks that the component responds correctly according to the design. In object-oriented programming, unit tests usually instantiate a class and call one of its methods. In an ASP.NET Core MVC web application, unit tests can instantiate model classes or controllers, and call their methods and actions. As a web developer, you need to know how to create a testing project, add tests to the testing project, and run tests in the testing project. The tests check individual aspects of your code without relying on database servers, network connections, and other infrastructure. This is because unit tests focus on the code you write.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of unit tests to improve the quality of code you write.
- List the principles of test-driven development (TDD).
- Describe how loosely-coupled components increase the testability and maintainability of your application.
- Write unit tests against model classes and controllers.
- Describe how dependency injection allows components to be more easily testable.
- Run unit tests in Microsoft Visual Studio 2017.
- Use a mocking tool to automate mock creation in unit tests.

### Why Perform Unit Tests?

There are several types of tests you can use to identify bugs in your application, which include:

- Unit tests. Unit tests check small aspects of functionality. For example, a unit test may verify the return type of a single method, or the method is returning the expected results. By defining many unit tests for your project, you can ensure they cover all functional aspects of your web application.
- Integration tests. Integration tests check how two or more components work together. You can use these tests to check how two or more classes interact with each other. You can also use integration tests to check how the entire web application, including the database and external web services, works to deliver content.
- Acceptance tests: Acceptance tests focus on a functional or technical requirement that must work for the stakeholders to accept the application. Similar to integration tests, acceptance tests usually test multiple components working together.

- Types of Tests:
  - Unit tests
  - Integration tests
  - Acceptance tests
- Unit tests verify that small units of functionality work as designed
  - Arrange. This phase of a unit test arranges data to run the test on
  - Act. This phase of the unit test calls the methods you want to test
  - Assert. This phase of the unit test checks that the results are as expected
- Any unit test that fails is highlighted in Visual Studio whenever you run the test or debug the application
- Once defined, unit tests run throughout development and highlight any changes that cause them to fail



**Note:** Unit tests are important because they can be defined early in development. Integration and acceptance tests are usually run later, when several components are approaching completion.

## What Is a Unit Test?

A unit test is a procedure that verifies a specific aspect of functionality. Multiple unit tests can be performed for a single class and even for a single method in a class. For example, you can write a test that checks that the **Validate** method always returns a Boolean value. You might write a second test that checks that when you send a string to the **Validate** method, the method returns a **true** value. You can assemble many unit tests into a single class called a test fixture. Often, a test fixture contains all the tests for a specific class. For example, **ProductTestsFixture** may include all tests that check methods in the **Product** class.

A single unit test usually consists of code that runs in three phases:

1. Arrange. In this phase, the test creates an instance of the class that it will test. It also assigns any required properties and creates any required objects to complete the test. Only properties and objects that are essential to the test are created.
2. Act. In this phase, the test runs the functionality that it must check. Usually, in the Act phase, the test calls a single procedure and stores the result in a variable.
3. Assert. In this phase, the test checks the result against the expected result. If the result matches what was expected, the test passes. Otherwise, the test fails.

## How Do Unit Test Help Diagnose Bugs?

Because unit tests check a small and specific aspect of code, it is easy to diagnose the problem when the tests fail. Unit tests usually work with a single class and isolate the class from other classes wherever possible. If other classes are essential, the smallest number of classes are created in the Arrange phase. This approach enables you to fix issues rapidly because the number of potential sources of a bug is small.

Unit tests should check the code that you write and not any infrastructure that the production system will rely on. For example, unit tests should run without connecting to a real database or web service because network problems or service outages may cause a failure. Using this approach, you can distinguish bugs that arise from code, which must be fixed by altering code, from the bugs that arise from infrastructure failures, which must be fixed by changing hardware, reconfiguring web servers, reconfiguring connection strings, or making other configuration changes.

By using dependency injection throughout the code instead of relying on explicit dependencies, the reliance on other classes and data sources can be bypassed, and it becomes possible to inject fake dependencies, allowing full testing of the class, without being hindered by external classes. Since the tested class expects interfaces, you can create fake dependent classes and implement various methods as needed with the expected results. This enables complete testing for a class, without reliance on other classes.

## Automated Unit Testing

It is important to understand that unit tests, after they are defined, can be rerun quickly and easily throughout the rest of the project life cycle. In fact, you can set up Visual Studio to rerun tests automatically whenever the related code changes. You can also manually initiate tests any time.

This is important because new code can cause bugs at any stage of the development process. As an example, consider a project that proceeds through three phases. A unit test defined for phase 1 helps highlight problems in phases 2 and 3 that might otherwise go unnoticed:

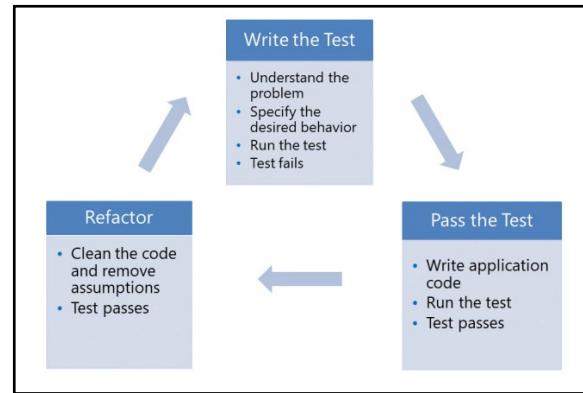
- Phase 1. The **ProductController** class is defined and the **Test\_Index\_Action** unit test is written. The test checks that the **Index** action works with an integer parameter. When you call the **Index** action with an integer, the action returns a View result action, that includes inside its model a collection that includes the same number of **Product** objects. The test passes.
- Phase 2. A developer modifies the **Index** action so that it returns a partial view. The developer renames the action **\_Index** to conform to the team's partial view naming convention. The test fails because the name has changed. The developer modifies the test so that it calls the renamed action and the test passes.
- Phase 3. A developer modifies the **Index** action by writing a different Language Integrated Query (LINQ) to implement a new functional requirement. However, the developer makes a mistake in the LINQ syntax. The **Index** action now returns a View result action with zero products on the model regardless of the integer that is passed as a parameter. The test fails.

In phase 2, the test failure arose because the action name had changed, not because of a mistake in the action code. The solution was to change the name of the action called in the test method, but this can remind developers to alter calls to renamed actions throughout the solution.

In phase 3, the test failure arose because the developer misunderstood LINQ or made a typing error in the LINQ syntax. The unit test highlighted the error as soon as it arose. This helps the developer focus on a solution to resolve the test failure and ensure that new functionality does not cause failures in code written earlier in the project.

## Principles of Test-Driven Development

To spot potential bugs and improve the quality of the final application, you can use unit tests in any development methodology, including waterfall projects, iterative projects, and agile projects. Any project can benefit from unit testing, regardless of the development methodology used in the project. However, a specific development methodology, called test-driven development (TDD), places unit testing at the center of working practices. TDD is often described as a separate development methodology. Some authors consider it to be a core principle of the extreme programming methodology.



 **Note:** Project development methodologies were covered in Module 2, "Designing ASP.NET Core MVC Web Applications".

### Write the Test, Pass the Test, Refactor

In TDD, developers repeat the following short cycle to implement a functional requirement:

- Write the Test. The developer starts by creating and coding the unit test. This step requires a full understanding of the functional requirement, which can be obtained from use cases or user stories. Because the developer has not written any code in the application, the test fails.
- Pass the Test. The developer writes some quick and simple code in the application so that it passes the test. During the first iteration, this code is frequently inelegant and may include false assumptions such as hardcoded values.

3. Refactor. The developer cleans up the application code, removes duplicate code, removes hardcoded values, improves readability, and makes other technical improvements. The developer reruns the test to ensure that refactoring has not caused a failure.

The cycle is then repeated. In each iteration, the developer adds a small new element of the final functionality with a corresponding test.

It is important that the code in your tests is treated as production code. It should be well thought out and easy to read so that other developers can understand the test and quickly diagnose any test failures.

### Test-Driven Development Principles

TDD is different from traditional approaches to application development. To use TDD effectively, you must understand its fundamental principles:

- Write tests before code. In the TDD development cycle, you write the test before writing any code in the application. This means the test must fail the first time it is run. You can understand the test as a specification for the functionality you are building. By writing the test first, you ensure that you begin with a thorough understanding of the problem you are trying to solve.
- Move in small steps. By breaking a large application down into small elements of functionality, you can improve developer productivity. You can do this by giving a developer a small problem to solve in each iteration. The developer can solve the simple problem quickly and understand all the possible circumstances in which their code runs.
- Only write enough code to pass the test. In each iteration, do not be tempted to add extra code that is not related to the test. For example, if you know that users will call an action with other parameters than the ones in the test, do not write code to handle these parameters. Instead, during the next iteration, write a second test for those parameters. Then write and refactor that code.

Developers can refer to tests as examples of how to use a class or method. This can increase developer productivity because developers can view the code that demonstrates the method or class.

## Writing Loosely Coupled MVC Components

A loosely coupled application is one in which each component requires few or no details of the other components of the system. For example, in object-oriented programming, two classes can be described as loosely coupled if one class can call methods on the other class without any code that is specific to the other class. When system components are loosely coupled in this manner, it is easy to replace a class with another implementation of the same functionality.

Loosely coupled components are essential for thorough unit testing because classes that deal with real data, such as data from a database, can easily be replaced with classes that deal with test data. When dependency injection is used inside an application, the application becomes more loosely coupled, since classes expect interfaces rather than classes.

- Loose coupling means that each component in a system requires few or no internal details of the other components in the system
- A loosely coupled application is easy to test because it is easier to replace a fully functional instance of a class with a simplified instance that is specifically designed for the test
- Loose coupling makes it easier to replace simple components with more sophisticated components
- Dependency injection inherently supports loose coupling

## Using Loose Coupling in Tests

A loosely coupled application is easy to test because you can make tests simpler by replacing a fully functional instance of a class with a simplified instance that is specifically designed for the test. Replacing classes in tests in this manner can only be done when components are loosely coupled. Replacement instances used for unit tests are known as test doubles or fakes. A test double or fake includes just enough code and properties to pass the relevant test and prove the functionality.

## Other Benefits of Loose Coupling

Loose coupling has other benefits besides testing. Loose coupling makes it easier to replace simple components with more sophisticated ones—for example, in version one of an application, you might write a class that calculates a simple arithmetical mean. In version two, you might replace this class with a new class that calculates a weighted mean based on a more complex algorithm. If the components are loosely coupled, you can perform this replacement without modifying any code outside of the averaging classes.

## Using Interfaces for Loose Coupling

In object-oriented programming, an interface defines a set of properties and methods. Any class that implements that interface must implement all the properties and methods it defines as a minimum. This creates loose coupling because you need not specify a class in code. Instead, you can specify any implementation of a particular interface.

## Loose Coupling in an MVC Web Application

To show how loose coupling can be used in an MVC web application, consider the following scenario:

You are writing an MVC web application that displays a product catalog and enables users to click a product to view the full details. On the product details page, users can view and add reviews of that product. You want to ensure during testing that the **Product.Reviews** property, which is a collection of **Review** objects, only includes reviews with the right **ProductID** value.

Recall that unit tests should not rely on infrastructure such as network connections or database servers, but only on test code. By contrast, in production, the application will obtain all product reviews from a database. To satisfy the needs of testing and production, you can build the following components:

- An **IProduct** interface. This includes a definition of the **Reviews** property, together with other properties and methods. This interface is known as a repository.
- A **Product** class. This is the implementation of the **IProduct** repository that the application uses in production. When the user calls the **Reviews** property, all the reviews for the current product are obtained from the database.
- A **TestProduct** class. This is the test double or fake implementation of the **IProduct** repository that the test uses. The test sets up a **TestProduct** object and fake reviews with different **ProductID** values. The test calls the **TestProduct.Reviews** property and checks that only the right reviews are returned.

Notice that, in the test, the **TestProduct** double uses in-memory data set up during the Arrange phase. It does not query the database. Therefore, you can test your code without relying on the network connection to the database or the database server itself. This approach also ensures that the test runs quickly, which is important because slow tests discourage developers from running tests regularly.

## Writing Unit Tests for MVC Components

The ASP.NET Core MVC programming model is easy to integrate with the principles of unit testing and TDD because of its separation of concerns into model, controllers, and views, as well as its support of dependency injection. Models are simple to test because they are independent classes that you can instantiate and configure during the Arrange phase of a test. Controllers are simple classes that you can test, but it is complex to test controllers with in-memory data, rather than using a database. To test controllers with in-memory data, you must create a test double class, also known as a fake repository. Objects of this class can be populated with the test data in memory without querying a database. You need to understand how to write test doubles and how to write a typical test for MVC classes.

- You can test an ASP.NET Core MVC web application project by adding a test project to the solution

- Model classes can be tested by instantiating them in-memory, arranging their property values, acting on them by calling a method, and asserting that the result was as expected

In this lesson, you will learn how to create and run unit tests by using the MSTest testing framework. MSTest is a framework that is built into Visual Studio and is maintained by Microsoft. Other popular choices are xUnit and NUnit, both of which can also be fairly easily used in Visual Studio.

### Adding and Configuring a Test Project

In Visual Studio, you can test an ASP.NET Core MVC web application project by adding a new project to the solution, based on the **MSTest Test Project (.NET Core)** template. You must add a reference from the test project to the MVC web application project so that the test code can access classes in the MVC web application project. You should also install the **Microsoft.AspNetCore.Mvc** package in the test project.

In a Visual Studio MSTest test project, test fixtures are classes marked with the **TestClass** attribute. Unit tests are methods marked with the **TestMethod** attribute. Unit tests usually return **void** but call a method of the **Assert** class, such as **Assert.AreEqual** to check results in the test Assert phase.



**Note:** There are many other test frameworks available for MVC web applications, and you can choose the one you prefer. Many of these frameworks can be added by using the NuGet package manager, which is available in Visual Studio.

### Test Model Classes and Business Logic

In MVC, model classes do not depend on other components or infrastructure. You can easily instantiate them in-memory, arrange their property values, act on them by calling a method, and assert that the result was as expected. Sometimes, you create business logic in the model classes, in which case, the Act phase will involve calling a method on the model class itself. If, by contrast, you have created a separate business logic layer, code in the Act phase must call a method on the business object class and pass a model class.

The following code is an example of a **Product** model, which will be tested:

#### Model for Testing

```
public class Product
{
 public string Name { get; set; }
 public int Id { get; set; }
 public float BasePrice { get; set; }
```

```
public float GetPriceWithTax(float taxPercent)
{
 return BasePrice + (BasePrice * (taxPercent / 100));
}
```

The following code is an example of a test for the **Product** model created previously:

### Testing a Model Class

```
[TestClass]
public class ProductTest
{
 [TestMethod]
 public void TaxShouldCalculateCorrectly()
 {
 // Arrange
 Product product = new Product();
 product.BasePrice = 10;

 // Act
 float result = product.GetPriceWithTax(10);

 // Assert
 Assert.AreEqual(11, result);
 }
}
```

This test is designed to check that the **GetPriceWithTax** method is working correctly.

### Testing Controller Classes

Unlike models, which are simple classes designed to be mainly self-sufficient without being reliant on other classes, controllers are more complex to test. To test controllers, you will need some additional work because an individual controller can potentially rely on models, repositories, and services and more.

The following code is an example of a **ProductController** class that needs to be tested:

### Controller for Testing

```
public class ProductController : Controller
{
 IProductRepository _productRepository;

 public ProductController(IProductRepository productRepository)
 {
 _productRepository = productRepository;
 }

 public IActionResult Index()
 {
 var products = _productRepository.Products.ToList();
 return View(products);
 }
}
```

### Creating a Repository Service

A repository is a common design pattern that is used to separate business logic code from data retrieval. This is commonly done by creating a repository interface that defines properties and methods that MVC can use to retrieve data. Usually, the repository handles the various CRUD (create, read, update, and delete) operations on the data. In ASP.NET Core MVC applications, the repository is handled by creating a service. In general, you will want one repository per entity. Repositories are prominently useful for testing controllers because models usually do not use external data.

The following code is an example of a repository interface:

### A Repository Interface

```
public interface IProductRepository
{
 IQueryable<Product> Products { get; }
 Product Add(Product product);
 Product FindProductById(int id);
 Product Delete(Product product);
}
```

### Implementing and Using a Repository in the Application

The repository interface defines methods for interacting with data but does not determine how that data will be set and stored. You must provide two implementations of the repository:

- A service implementation of the repository that will be used inside the application. This implementation will use data from the database or some other storage mechanism.
- An implementation of the repository for use in tests. This implementation will use data from the memory set during the Arrange phase of each test.

The following code example shows a service implementation of a repository:

### Implementing a Repository in the ASP.NET Core MVC Project

```
public class ProductRepository : IProductRepository
{
 StoreContext _store;
 public ProductRepository(StoreContext store)
 {
 _store = store;
 }

 public IQueryable<Product> Products
 {
 get { return _store.Products; }
 }

 public Product Add(Product product)
 {
 _store.Products.Add(product);
 _store.SaveChanges();
 return _store.Products.Find(product);
 }

 public Product Delete(Product product)
 {
 _store.Products.Remove(product);
 _store.SaveChanges();
 return product;
 }

 public Product FindProductById(int id)
 {
 return _store.Products.First(product => product.Id == id);
 }
}
```

Note that the **StoreContext** is a class which inherits from the Entity Framework **DbContext** class, which is injected through dependency injection and declared inside of the **ConfigureService** method. In fact, the interface methods such as **Add**, **Delete**, and **FindProductById** just wrap methods from the **DbContext** class such as **Remove**.



**Note:** Working with the **DbContext** class was covered in Module 7, "Using Entity Framework Core in ASP.NET Core".

## Implementing a Repository Test Double

The second implementation of the repository interface is the implementation that you will use in unit tests. This implementation uses in-memory data and a keyed collection of objects to function just like an Entity Framework context but avoids working with a database.

The following code example shows how to implement a repository class for tests:

### Implementing a Fake Repository

```
public class FakeProductRepository : IProductRepository
{
 private IQueryable<Product> _products;

 public FakeProductRepository()
 {
 List<Product> products = new List<Product>();
 _products = products.AsQueryable();
 }

 public IQueryable<Product> Products
 {
 get { return _products.AsQueryable(); }
 set { _products = value; }
 }

 public Product Add(Product product)
 {
 List<Product> products = _products.ToList();
 products.Add(product);
 _products = products.AsQueryable();
 return product;
 }

 public Product Delete(Product product)
 {
 List<Product> products = _products.ToList();
 products.Remove(product);
 _products = products.AsQueryable();
 return product;
 }

 public Product FindProductById(int id)
 {
 return _products.First(product => product.Id == id);
 }
}
```

## Using the Test Double to Test a Controller

After you have implemented a test double class for the repository, you can use it to test a controller in a unit test. In the Arrange phase of the test, create a new object from the test double class and pass it to the controller constructor. The controller uses this object during the test Act phase. This enables you to check results.

The following code example shows how to test a controller action by using a test double:

### Using a Test Double to Test a Controller

```
[TestClass]
public class ProductControllerTest
{
 [TestMethod]
 public void IndexModelShouldBeListOfProducts()
 {
 // Arrange
 var productRepository = new FakeProductRepository();
 productRepository.Products = new { new Product(), new Product(), new Product() }
 .AsQueryable();
 var productController = new ProductController(productRepository);

 // Act
 var result = productController.Index() as ViewResult;

 // Assert
 Assert.AreEqual(typeof(List<Product>), result.Model.GetType());
 }
}
```

Note that the test creates a new instance of the **FakeProductRepository** class and adds three **Product** objects to it. In this case, you can run the test without setting any properties on the **Product** objects. This context is passed to the **ProductController** constructor and the test proceeds to the Act and Assert phases.

### Running Unit Tests

Once you write a test, you can run it by right-clicking a test class, and then selecting the **Run Test(s)** option in the context menu. Alternatively, you can open the test explorer from the test menu inside the windows sub options. From the test runner explorer, you can then choose specific test classes or methods and run them as needed. There are also options to run all tests, or run tests while in debug mode, allowing you to observe specific failures and handle them accordingly.

 **Note:** When developing an ASP.NET Core application, it is possible to run Live Unit Testing. This allows you to run unit tests as you code, ensuring the integrity of your code without having to constantly run tests manually. You can find out more about Live Unit Testing here:  
<https://aka.ms/moc-20486d-m10-pg1>

## Demonstration: How to Run Unit Tests

In this demonstration, you will learn how to add a new test project to a solution to test an ASP.NET Core MVC application. After that, you will learn how to write unit tests and run them.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Run Unit Tests" on the following page:

<https://github.com/MicrosoftLearning/20486D->

[DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD10\\_DEMO.md#demonstration-how-to-run-unit-tests.](DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD10_DEMO.md#demonstration-how-to-run-unit-tests.)

## Using Mocking Frameworks

When you write a unit test, you must, during the Arrange phase, create test data on which the test will run. By using a test double or mock object to supply test data, you can test your code in isolation from other classes and from the infrastructure elements such as databases and network connections on which the application will rely. You can create test doubles by manually coding their instantiation, setting their properties, and populating collections. Such test doubles are known as manual mock objects.

- A mocking framework automates the creation of mock objects during tests
  - You can automate the creation of a single object
  - You can automate the creation of multiple objects of the same type
  - You can automate the creation of multiple objects that implement different interfaces
- The mocking framework saves time when writing unit tests

Alternatively, instead of creating mock objects manually with your own code, you can use a mocking framework to automate this work. A mocking framework automatically creates a mock object by using the interfaces that you specify. In the case of IoC containers, there are many mocking frameworks that can be used in testing MVC web applications. You can find many mocking frameworks by using the NuGet package manager.



**Note:** Popular mocking frameworks for ASP.NET Core MVC include Moq and NSubstitute. These frameworks are available in NuGet and there are additional alternatives. Choose the framework that best suits your testing needs.

### Why Use a Mocking Framework?

There are many situations in which mocking frameworks can significantly ease unit testing. Even for simple tests, mocking frameworks reduce the amount of setup code that you have to develop. After you become familiar with the mocking framework that you choose and have learned how to write arrangement code for it, you will begin to save time. In more complex situations, such as the following, mocking frameworks have great advantages:

- Creating multiple mock objects of a single type. You should try to keep each unit test as simple as possible, but inevitably, some tests require many mock objects to work with. For example, if you are testing a method that underlies paging functionality in your web application, you must create enough mock objects to populate multiple pages. Mocking frameworks can automate the creation of multiple mock objects of the same type.
- Mocking objects with multiple interfaces. In some tests, where there are several dependencies between classes of different types, you must create many mock objects of different classes. In such situations, it is tedious to manually code many mock objects. Mocking frameworks can help by automatically generating the objects from the interfaces that you specify.

In each unit test, you are interested in a small and specific area of functionality. Some properties and methods of an interface will be crucial to your test, while others will be irrelevant. A mocking framework enables you to specify irrelevant properties and methods in a given test. When the framework creates a mock object for your test, it creates stubs for the irrelevant properties and methods. A stub is a simple implementation with little code. In this way, the mocking framework frees you from having to implement all the requirements of the interface laboriously.

The following code is an example of a controller unit test by using the Moq framework:

### Moq Framework Unit Test

```
[TestClass]
public class ProductControllerTest
{
 [TestMethod]
 public void IndexModelShouldBeListOfProducts()
 {
 // Arrange
 var productRepositoryMock = new Mock<IProductRepository>();
 productRepositoryMock.SetupGet(repos => repos.Products).Returns(new[] { new
Product(), new Product(), new Product() }.AsQueryable());
 var repository = productRepositoryMock.Object;
 var productController = new ProductController(repository);

 // Act
 var result = productController.Index() as ViewResult;

 // Assert
 Assert.AreEqual(typeof(List<Product>), result.Model.GetType());
 }
}
```

In this example, you can see a unit test that leverages the strengths of a mocking framework. Unlike the previous controller test, in this test, you do not need to create a testing double for `IProductRepository`. Instead, by using the mock object exposed by the Moq framework, you can set up a fake class that returns a list of products that you defined.

## Lesson 2

# Implementing an Exception Handling Strategy

Unexpected events are likely to occur from time to time in any complex system, including MVC web applications. Occasionally, such unexpected run-time events cause an error. When this happens, ASP.NET Core or the .NET Framework generates an exception, which is an object that you can use to diagnose and resolve the error. The exception contains information that you can use to diagnose the problem. Exceptions that are not handled in your code will cause the web application to halt and an error message to be displayed to the user. As a web developer, you need to know how to detect, handle, and raise exceptions, and identify the cause of the problem. Visual Studio provides a broad range of tools for debugging exceptions and improving the robustness of your code.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to raise and catch exceptions.
- Run your application in multiple environments.
- Utilize middleware to handle exceptions in an ASP.NET Core MVC Application.
- Configure error handling in an ASP.NET Core application.

### Raising and Catching Exceptions

An error is an unexpected run-time event that prevents an application from completing an operation. When a line of code causes an error, ASP.NET Core or the common language runtime (CLR) creates an exception. This exception is an object of a class that inherits from the **System.Exception** base class. There are many exception classes. Often the object class identifies what went wrong. For example, if there is an **ArgumentNullException**, it indicates that a **null** value was sent to a method that does not accept a **null** value as an argument.

- The most common method to catch an exception is to use the **try/catch** block
- You can add custom exceptions or use existing ones

```
throw new ArgumentNullException();
...
try
{
 price = product.GetPriceWithTax(-20);
}
catch (InvalidTaxException ex)
{
 return Content("Tax cannot be negative");
}
```

### Unhandled Exceptions

When an exception is not explicitly handled by an application, the application stops, and the user sees an error message. In ASP.NET Core MVC applications, this error message is in the form of a webpage. You can override ASP.NET Core default error pages to display your own error information to users.

If an unhandled exception arises while you are debugging the application in Visual Studio, execution breaks on the line that generated the exception. You can use the Visual Studio debugging tools to investigate what went wrong, isolate the problem, and debug your code.

Sometimes, you may also want to raise your own exceptions to alert components that something in your application went wrong. For example, consider an application that calculates prices for products after adding taxes. In the **Product** model you implement a **GetPriceWithTax** method that adds a tax percentage to a product's base price. If the tax percentage is a negative number, you may wish to raise an exception. This approach enables code in the controller that calls the **GetPriceWithTax** method, to handle the event that a negative tax percentage was used. You can use the **throw** keyword to raise errors in C# code.

## Catching Errors with Try/Catch Blocks

The most commonly used way to catch an exception, which works in any Microsoft .NET Framework code, is to use the **try/catch** block. Code in the **try** block is run. If any of that code generates an exception, the type of exception is checked against the type declared in the **catch** block. If the type matches or is of a type derived from the type declared in the **catch** block, the code in the **catch** block runs. You can use the code in the **catch** block to obtain information about what went wrong and resolve the error condition.

The following code example demonstrates a try/catch block that catches an **ArgumentNullException**:

### A Simple Try/Catch Block

```
try
{
 Product product = FindProductFromComment(comment);
}
catch (ArgumentNullException ex)
{
 // Handle the exception
}
```

In this example, you will catch any exception of the **ArgumentNullException** type. If an exception occurs of a type other than **ArgumentNullException**, the exception will not be caught.

## Creating Custom Exception Types

While there are many types of exceptions built in to the system, sometimes you may need to raise an exception that does not fit any of the standard ones. For this purpose, you can create a custom exception class. This can allow you to create exceptions that make sense for your application.

In order to create a custom exception, you will need to create a new class that inherits from the class **Exception**, or any other class that derives from **Exception**. Doing this will allow you to use the class to throw exceptions, allowing for more specific exception handling in your application.

The following code is an example of a custom exception class:

### Custom Exception Class

```
public class InvalidTaxException : Exception
{
}
```

The following code is an example of a method that throws a custom exception:

### Throw Custom Exception

```
public float GetPriceWithTax(float taxPercent)
{
 if (taxPercent < 0)
 {
 throw new InvalidTaxException();
 }
 return BasePrice + (BasePrice * (taxPercent / 100));
}
```

The following code is an example of catching a custom exception:

### Catching a Custom Exception

```
try
{
 float price = product.GetPriceWithTax(-20);
}
catch (InvalidTaxException ex)
{
 return Content("Tax cannot be negative");
}
```

 **Best Practice:** It is considered best practice to always end exception classes with the suffix **Exception**. This lets the developer immediately understand why this class exists and how to use it correctly.

## Working with Multiple Environments

In most applications, there is a distinct requirement for separate behaviors depending on whether the application is run in the developer's computer or being used to serve clients on a production server. While developing, you will need to diagnose complex error messages, debug JavaScript code, ensure that you are working with the latest versions of files, and use frequent logging for the tracing of errors. However, on the production environment, the user experience should be prioritized for speed and performance. In production environment, compressed JavaScript code should be used. Having the client browser download the same files every visit can cause clients to become dissatisfied by the speed of your application. Furthermore, giving clients detailed exception details will cause an unpleasant experience for the average user, and allow malicious users to know more details about your application.

- Use the environment variable **ASPNETCORE\_ENVIRONMENT** to determine application environment
- The **IHostingEnvironment** interface exposes useful methods:
  - **IsDevelopment**
  - **IsStaging**
  - **IsProduction**
  - **IsEnvironment(\*Environment Name\*)**

In ASP.NET Core, the environment variable **ASPNETCORE\_ENVIRONMENT** should be used to determine application environment for all ASP.NET Core applications. If not set, the default value of the **ASPNETCORE\_ENVIRONMENT** environment variable is **Production**. This ensures that any application which is run on an environment without the specification will always run in **Production** mode. You can, at any time, add it to your environment, and give it other values. By default, **Development**, **Staging**, and **Production** are all commonly used values. However, you can also add any possible string value you wish, since you may want a specific environment for unit testing, multiple production environments, and so on.

In an ASP.NET Core application, particularly while declaring middleware, it is possible to use the current runtime environment to make decisions which affect your application's behavior. A very common usage is for exception handling. While running an application in **Development** mode, you will often want error pages to contain detailed stack traces with as much information as possible about the exception that occurred. However, when doing so in a production environment, it will cause clients to receive entire pages full of error details which mean nothing and are undesirable for the majority of users. These pages contain information that you must not expose to malicious users to potentially exploit.

In order to resolve this, you can use the **IHostingEnvironment** service. This service is frequently injected into the **Configure** method of the **Startup** class. The **IHostingEnvironment** interface exposes useful methods in the form of **IsDevelopment**, **IsStaging**, **IsProduction**, and **IsEnvironment(\*Environment Name\*)**. You can use these methods to determine the currently running environment.

The following code is an example of using the **IHostingEnvironment** service:

### Using the IHostingEnvironment Service

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 if (env.IsDevelopment())
 {
 app.UseDeveloperExceptionPage();
 }
 else if (env.IsStaging() || env.IsProduction() ||
env.IsEnvironment("ExternalProduction"))
 {
 app.UseExceptionHandler("/error");
 }

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("...");
 });
}
```

In this example, you can see that in a **Development** environment, the **UseDeveloperExceptionPage** method is called, and in **Staging**, **Production**, and **ExternalProduction** environments, the **UseExceptionHandler** with a route of **/error** is called instead. The **UseDeveloperExceptionPage** and **UseExceptionHandler** methods will be covered in the third topic of this lesson, "Configuring Error Handling".

### Methods of Startup Class for Specific Environment

It is also possible to create specific **Configure** and **ConfigureServices** methods for a specific environment. This can be done by adding the environment name at the end of the method name. It is important to note that using this will override the default **Configure** or **ConfigureServices** behavior. Therefore, you should avoid relying on this too often. As an example for a good use case, if you want to test specific middleware during development, these methods will allow you to run that middleware and focus on debugging without possibly breaking production environments.

The following code is an example of an environment specific **Configure** method:

### Environment Specific Configure Method

```
public void ConfigureExternalProduction(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Welcome to external production");
 });
}
```

Note that, in this example, **Configure** is overridden in the custom **ExternalProduction** environment.

## Using Environments in Views

Another location in which it can be useful to differentiate between environments can be inside views, particularly when setting up the application. You will often want to use uncompressed versions of JavaScript and CSS files while running in a development environment, compared to a production environment where you will want to use bundled and minified files instead. In order to achieve this, you can use the **environment** tag helper. The **environment** tag helper allows you to use an **environment** tag, and by setting the **include** or **exclude** property on it, you are able to specify in which environments the code will be run. This is particularly useful for loading scripts and CSS files.

The following code is an example of using the **environment** tag helper:

### Environment Tag Helper

```
<!DOCTYPE html>
<html>
<head>
 <environment include="Development">
 <link rel="stylesheet" href("~/Styles/bootstrap.css" />
 </environment>
 <environment exclude="Development">
 <link rel="stylesheet" href("~/Styles/vendor-min.css" />
 </environment>
</head>
<body>
 <div class="container body-content">
 @RenderBody()
 </div>
 <environment include="Development">
 <script src "~/Scripts/jquery.js"></script>
 <script src "~/Scripts/popper.js"></script>
 <script src "~/Scripts/bootstrap.js"></script>
 </environment>
 <environment include="Production,Staging">
 <script src "~/Scripts/vendor.min.js"></script>
 </environment>
</body>
</html>
```

In this example, you can see that while in **Development** mode, the application will use uncompressed JavaScript and CSS files. In **Production** and **Staging**, the application will serve bundled and minified files instead.



**Note:** The **environment** tag helper is located in the **Microsoft.AspNetCore.Mvc.TagHelpers** namespace. To use it in a view, you need to add the **@addTagHelper** directive to the view to define the tag helpers that this view will use. Alternatively, you can add the **@addTagHelper** directive to a **\_ViewImports.cshtml** file that is located under the **Views** folder, so that it is available to all the views. Tag helpers were covered in Module 5, "Developing Views".

## Using Environments in Configuration Files

There are many additional places in the application, in which differentiating environments is both simple and useful. A common example is the **appsettings.json** file. By using the following file naming scheme, **appsettings.\*environmentName\*.json**, you can create environment-specific configurations. Using this to support multiple separate connection strings is particularly useful because a production database will usually have separate databases. An **appsettings.json** file for a specific environment will always take priority over the general **appsettings.json** file.



**Note:** The **appsettings.json** file was described in Module 7, "Using Entity Framework Core in ASP.NET Core".

## Using Profiles

While most servers run on a single consistent environment, however, in a development environment that is not the case. As a developer, you will often be tasked with checking a variety of issues that can be reproduced only on specific environments, and you will often end up switching environments many times. In order to facilitate this, you can set up various development environments within Visual Studio. To configure environments for your project, right-click the project file, select **properties** from the menu, and then select the **Debug** menu.

Using Visual Studio, you can create a new runtime profile for the application. Each profile has a unique name. In addition, each profile has a **commandName** property, which is used to determine the web server to launch. Common values for the **commandName** property include IIS Express, which specifies that IIS Express will be launched, and **Project**, which specifies that Kestrel server will be launched. The differences between IIS Express and Kestrel will be covered further in Module 14, "Hosting and Deployment". In the profile, you can also add and change environment variables. These variables determine various behaviors at run time, and of particular importance is the **ASPNETCORE\_ENVIRONMENT** variable. By setting the value of this key, you can set the environment in which the application is run. In a profile, you can also specify if a browser will open on your application whenever you run it from inside Visual Studio.

Note that after you have added a profile, it can also be seen inside of the project's **Properties** section inside of the **launchSettings.json** file. You can also directly modify the **launchSettings.json** file yourself to create new developments profiles.

To run the development profile, you can use the debugging menu in the Visual Studio toolbar. By clicking the arrow next to the run button, a drop-down list will appear. From this list, you can select the development profile. You can then run the application using the selected profile. This allows you to switch development environments on the fly, allowing for a fast response time and the ability to test environment changes.

Another easy way in which you can run your application in various environments is by using the command line to set the **ASPNETCORE\_ENVIRONMENT** variable, and then execute the **dotnet run** command to run the application. The **dotnet run** command will run an application by using the first profile inside of the **launchSettings.json** file that uses the **commandName** property of the project. By setting **ASPNETCORE\_ENVIRONMENT** beforehand the project will run by using the environment set in the **ASPNETCORE\_ENVIRONMENT**. This configuration will persist only within that specific command line window. Note that this will only work if the **ASPNETCORE\_ENVIRONMENT** variable is not specifically set inside the profile configuration. The **dotnet run** command will be covered in module 14, "Hosting and Deployment".

The following code is an example of running the application by using the **dotnet run** command:

### Using the **dotnet run** Command

```
set ASPNETCORE_ENVIRONMENT=Production
dotnet run
```



**Note:** Note that the setting of the **ASPNETCORE\_ENVIRONMENT** variable in Linux and macOS may be slightly different, but the behavior will be the same.

## Configuring Error Handling

In ASP.NET Core MVC, you will need to be able to handle various errors that occur. In this topic, you will learn how to set up the application to create user friendly error pages that can be displayed to the clients in the production environment, as well as how to show detailed error logs which developers can use to trace problems. In addition, you will also learn how to handle specific cases such as controller specific errors.

In a development environment, you will often want detailed information about errors. This can allow you to extrapolate important information about what went wrong and how. By adding the **UseDeveloperExceptionPage** middleware to the pipeline inside the **Configure** method, all exceptions that occur in middleware after the **UseDeveloperExceptionPage** middleware will be redirected by default to a page that has detailed information about the exception. Inside the page, you can switch between various tabs to see important information such as the stack trace of the exception, the query string data, the cookies data, and the headers sent as part of the request. This can all be extremely helpful while debugging the application and for reproducing known errors because the relevant information for the exception is present and easily accessible.

The following code is an example of using the **UseDeveloperExceptionPage** middleware:

### The **UseDeveloperExceptionPage** Middleware

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 if (env.IsDevelopment())
 {
 app.UseDeveloperExceptionPage();
 }

 app.Run(async (context) =>
 {
 await Task.Run(() => throw new NotImplementedException());
 });
}
```

In ASP.NET Core applications, there are many ways to handle errors including:

- Using the developer exception page
- Using an exception handler to direct to a custom error page
- Using status code pages
- Using exception filters to catch exceptions in specific actions and controllers



**Best Practice:** The **UseDeveloperExceptionPage** middleware can expose many details about the application structure. Therefore, it should only be used in a development environment.

### Custom Exception Handler Page

While in a development environment, you will often want detailed exceptions and errors. In other environments, you will want to use more user-friendly and immediately helpful pages. For that purpose, you can use the **UseExceptionHandler** middleware. This middleware receives a string URL to which the request will be redirected. This URL can be used to redirect the problematic request to a controller that handles exceptions by providing a helpful user-friendly view, or to a static HTML file.

The following code is an example of using a custom exception handler:

### The **UseExceptionHandler** Middleware

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 if (env.IsDevelopment())
 {
 app.UseDeveloperExceptionPage();
 }

 else if (env.IsStaging() || env.IsProduction() || env.IsEnvironment("ExternalProduction"))
 {
 app.UseExceptionHandler("/error");
 }

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await Task.Run(() => throw new NotImplementedException());
 });
}
```

Note that, in order to implement a controller as an exception handler, you will need to call the **UseMvc** or the **UseMvcWithDefaultRoute** middleware. Alternatively, if you prefer using a static HTML file, you can use an exception handler that redirects to a static file alongside the **UseStaticFiles** middleware.

As a general rule, you will want to keep this page as static as possible since if an error handling controller throws exceptions, a generic server error page will be displayed instead. While creating controllers and views for displaying errors, ensure that you keep the logic simple.

The following code demonstrates a controller that functions as an exception handler:

### Exception Handler Controller

```
public class ErrorController : Controller
{
 public IActionResult Index()
 {
 return View();
 }
}
```

 **Best Practice:** Avoid using HTTP method attributes inside error handlers because the request might be of different types, which could result in exceptions not getting handled.

### Status Code Pages

An additional method of handling errors at your disposal is the **UseStatusCodePages** middleware. This middleware is designed to provide basic handling for various errors that may occur by displaying a short explanation of the error code to the user. In general, the **UseStaticCodePages** middleware is designed to handle cases where some part of the request went wrong without throwing an exception. This method will capture html status codes between 400 to 599, which generally cover all cases where something went wrong (400s represent errors on the client side, while 500s cover server-side errors).

The following code is an example of using the **UseStatusCodePages** middleware:

### Using the UseStatusCodePages Middleware

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseStatusCodePages();

 app.Run(async (context) =>
 {
 await Task.Run(() => context.Response.StatusCode = 404);
 });
}
```

You are not limited to calling just the **UseStatusCodePages** middleware for handling HTTP status codes. The following methods are also available:

- **UseStatusCodePagesWithRedirects**. Allows sending a redirection URL to the client. This new URL will be used instead of the original one. Using the *{0}* parameter in the URL can be used to set status codes.
- **UseStatusCodePagesWithReExecute**. Similar to redirection. However, this redirection occurs directly in the pipeline without affecting the client.

### Handling Exceptions in the Server

Sometimes, in most ASP.NET Core MVC applications, an exception might occur on the web server, instead of inside your ASP.NET Core MVC application code, and not be caught inside the application, but caught by the server instead. In this case, if the exception is caught before the response headers are sent, a response will be sent with no body and a 500 status code (internal server error). On the other hand, if the response headers were sent before the server catches the exception, the connection is immediately closed by the server. It is therefore important to avoid any exceptions leaving your application because such errors can be very difficult to trace.

### Handling Model State Errors

Another option for handling exceptions is through the model. As part of the action of the controller, the model is validated as is covered in Module 6, "Developing Models". By using the **IsValid** property on the model inside the controller action, you can stop further processing invalid data and take appropriate actions on the controller.

### Using Exception Filters

You can also configure a special filter called exception filter (Filters were previously covered in Module 4, "Developing Controllers"). You can create an exception filter by creating a class that inherits from the **ExceptionFilterAttribute** class. This filter can then be applied as an attribute to specific actions and controllers, which allows for handling specific exception cases.

The class that implements the exception filter class should override the **OnException** method inside it. You can then use the **ExceptionContext** parameter to retrieve or set the current exception, you can update the **Result** property, potentially changing the view or the response context, and you can also set the **ExceptionHandled** property to stop the propagation of the exception (If it is left as false, the exception will keep propagating).

The following code is an example of an exception filter:

#### Exception Filter Class

```
public class MyExceptionFilter : ExceptionFilterAttribute
{
 public override void OnException(ExceptionContext context)
 {
 if (!context.ModelState.IsValid)
```

```

 {
 var result = new ViewResult { ViewName = "InvalidModel" };
 context.Result = result;
 context.ExceptionHandled = true;
 }
}
}

```

The following code is an example of applying an exception filter to an action:

### Action with Exception Filter

```

public class ProductController : Controller
{
 [MyExceptionFilter]
 public IActionResult Index(Product product)
 {
 if (!ModelState.IsValid)
 {
 throw new Exception();
 }
 return View(product);
 }
}

```

In this example, you can see an error filter that redirects to the **InvalidModel** view whenever an exception occurs with an invalid model.



**Best Practice:** Note that as a general rule, creating error handling middleware is preferable, and exception filters are best left for edge cases.

## Demonstration: How to Configure Exception Handling

In this demonstration, you will see how to find and solve exceptions that are raised in an ASP.NET Core application. During the investigation of the exceptions, the environment will be changed from Production to Development. After the errors are fixed, the environment will be changed from Development to Production.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Configure Exception Handling” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD10\\_DEMO.md#demonstration-how-to-configure-exception-handling](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD10_DEMO.md#demonstration-how-to-configure-exception-handling).

## Lesson 3

# Logging MVC Applications

Throughout an application's life span, unexpected exceptions might occur on a daily basis. While it is usually reasonably easy to find and debug on development environments, the same cannot be said about production environments. In most cases, as a developer, you will not be able to directly work in the production environment. In order to make it easier to find bugs, you need to add regular logs throughout the application to monitor the flow of the application and point out exceptions. Doing this can help you find issues in your applications and deal with bugs that were not caught before production.

### Lesson Objectives

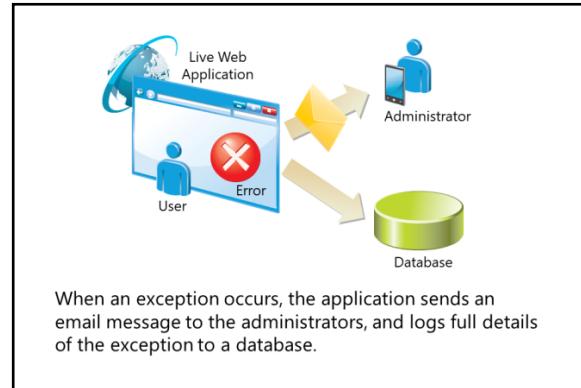
After completing this lesson, you will be able to:

- Decide the type of logging to be used in your application.
- Add logging providers to allow your application to log exceptions and important information as it runs in a specified manner.
- Call logging methods throughout your application to log important events.

### Logging Exceptions

Exceptions that you face during development can be investigated and debugged by using the Visual Studio debugging tools. In an ideal situation, no exceptions would arise when your web application is complete and deployed for users over the Internet. However, in the real world, unforeseen circumstances arise resulting in exceptions. For example, database failures, network issues, and configuration errors in any part of the system can cause exceptions.

You have already seen how to present a branded and informative custom error page to users when exceptions occur. It is also appropriate to log the exceptions in a production web application, so the administrators and developers can assess the extent of the problem, remove the cause, and improve the robustness of the code.



### Writing Error Logging Code

When adding logs to your application, you need to first decide where you would like to store the logs. There are many potential options you can consider for your application, and no single solution is the right one because each solution has its pros and cons. Popular logging methods include writing log files with various details about the application, sending logging data to cloud-based services for ease of storage and analysis, storing logging data in databases, and sending e-mails about exceptions.

By storing logs on files, you ensure that the logs are persistently written, and most reliably maintained. However, care must be taken to balance the logs to maintain a coherent way of tracking the flow of data and exceptions because logging too much can result in large, hard-to-use files that take up valuable space.

By storing logs on cloud-based services such as Microsoft Azure, you can ensure easy access and powerful research tools for developers. However, cloud-based services rely on constant internet access.

By storing logs on a database, you can store and divide the logs in a way that can make them easy to peruse. However, this requires a steady connection to the database and appropriate database administration. Also, these logs may not be easily accessible to the developers on a daily basis.

By logging to email, you can ensure that developers are informed very quickly and can handle the problem almost as soon as it occurs. However, it can also easily result in repeated emails for repeated issues, an excess of emails being sent, losses over network connections, and a difficult-to-manage email inbox.

Ensure that you discuss these options and any additional options with your team and decide which option is most appropriate for your application. Remember that you can use more than one method if it fits your application. For example, you could log to files on a daily basis while also mailing developers whenever critical exceptions occur.

### Where to Write Error Logging Code

When you decide where to write code that logs errors, you should consider that errors might arise in almost any part of your application. You should choose an approach that enables you to write error logging code once that will run for any exception anywhere in your application.

For example, it is not appropriate to write error logging code in individual **try/catch** blocks. If you do this, you will have to create a **try/catch** block for every procedure in your application and write duplicate logging code in the **catch** block.

A far more effective approach is to log by using middleware. By logging exceptions by using middleware, you can ensure that problematic code is properly logged and traced. This allows for a singular point of failure to handle all errors in the application.

### Using Third-Party Logging Tools

Because exception logging is a very common functional requirement for web applications, there are many third-party solutions that you can choose if you do not want to write your own logging code. Many of these are available within Visual Studio from the NuGet package manager.

## Logging in ASP.NET Core

In ASP.NET Core, there is a robust built-in logging library. Rather than working with a specific form of logging, using the logging library, you can setup multiple different logging providers, including built-in providers and various supported third-party logging providers. This allows you to implement the logging that is best for your application and update your logging mechanism without making any changes to the rest of your code.

### Configuring Logging Providers

The first step for setting up your logging is by setting up the providers you wish to use. This is done inside the **CreateWebHostBuilder** method of the **Program.cs** file. The **Program.cs** file and the **CreateWebHostBuilder** method will be covered in Module 14, "Hosting and Deployment".

```
public IActionResult Index()
{
 _logger.LogDebug("Index controller was entered");
 try
 {
 int x = 3;
 x -= 3;
 int result = 30 / x;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "An error occurred while dividing!");
 }
 return Content("Result from controller");
}
```

To add logging, you will need to add a call to the **ConfigureLogging** method as part of the **CreateDefaultBuilder** pipeline. The **ConfigureLogging** method exposes two parameters, a parameter of the **WebHostBuilderContext** type, which can be used to retrieve the **IHostingEnvironment**, and an **ILoggingBuilder** parameter, which is used to set up the logger providers and configurations.

In the **ConfigureLogging** method, you will need to add the logging provider, which will enable logging by using the specific provider. You can log to more than one provider at the same time. ASP.NET Core has several built-in providers, which include:

- **Console**. Logs the message to the application's console window.
- **Debug**. Logs the message to the application's debug window.
- **EventSource**. Uses event tracing API to store the event, behavior differs between operating systems, and it may not work on all platforms.
- **EventLog**. Logs the message to the windows event log. Is exclusive to Windows.
- **TraceSource**. Creates a trace source that can be listened to with a variety of trace listeners.
- **AzureAppServices**. Creates logs that integrate with the Azure platform.

The providers can be added by calling the **add\*Provider Name\*()** method. After you call this method, the provider is set up for logging throughout the application.

The following code is an example of using the **ConfigureLogging** method:

### The ConfigureLogging Method

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
WebHost.CreateDefaultBuilder(args)
 .ConfigureLogging((hostingContext, logging) =>
{
 if (hostingContext.HostingEnvironment.IsDevelopment())
 {
 logging.AddConsole();
 }
})
 .UseStartup<Startup>();
```

In this example, you can see that the application will write logs to the console. While running the application from Visual Studio, this can be found in the **Output** tab under **ASP.NET Core Web Server**.

### Calling Logging Method

After the logging is set up, you can add logs to individual components by injecting a logger into the constructor for the component. The logger that you inject should be of the **ILogger<\*className\*>** type where the class name is used to create an identifier for the file from which the logger is called. This identifier contains the fully qualified namespace for the class and makes it easy for developers to find out which log entry was created from where.

The following code is an example of a controller which writes to a log file:

### Logging from Controller

```
public class HomeController : Controller
{
 private ILogger _logger;

 public HomeController(ILogger<HomeController> logger)
 {
 _logger = logger;
 }

 public IActionResult Index()
```

```

 {
 _logger.LogInformation("Adding an entry to the logger.");
 return Content("Result from controller");
 }
}

```

The following code is an example of using logging in the **Configure** method:

### Logging from the Configure Method

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILogger<Startup>
 logger)
{
 logger.LogCritical("Critical Message from logger");

 if (env.IsDevelopment())
 {
 app.UseDeveloperExceptionPage();
 }

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Hello World!");
 });
}

```

 **Best Practice:** While it is possible to give the **ILogger** any class, it could result in logs becoming confusing to use. It is recommended to always create a logger for the current class.

### Log Levels

In an ASP.NET Core MVC application, you will often want to keep track of a wide variety of events. At times, you might want to keep track of every individual DB request or every single call to the server, while in other times you might want to track major flows being called and exceptions that occur. For this purpose, you can use log levels to control which logs will be displayed in your application. By setting a log severity, only logs that match the severity or of a higher severity will be displayed.

The log levels and their severity values are as follows:

- **Trace** (0). These should be used when trying to debug specific things and are often used to track very large amounts of data. By default, trace logs are never displayed and should not be used outside of a development environment.
- **Debug** (1). Logs of this level are frequently used for development environments and keep track of large amounts of data. Logs of this level should not be logged outside of development environments.
- **Information** (2). Logs of this level should be used at important points throughout the application to ensure the application is running correctly but should avoid getting to a point where they affect system performance. Information logs are frequently, but not always, enabled in production environments.
- **Warning** (3). Logs of this level should be used whenever unexpected flows occur. Exceptions that are handled or flows with missing data will often log a warning. These logs should always appear in a development environment, which allows you to find problematic flows.
- **Error** (4). Logs of this level should occur whenever exceptions cannot be handled. These should be cases in which a flow has broken down but did not result in application wide issues. These logs should always be kept.

- **Critical** (5). Logs of this level should be used to signify application wide failures. Failures such as an invalid database connection and a failure on the web server should be tracked with critical logs.

To call logs of different levels, you can use the **Log\*level\*** method. For example, to create an information level log, you should use **LogInformation**, and in order to create a critical log, you should use **LogCritical**.

The following code is an example of log levels:

### Log Levels Example

```
[HttpPost]
public IActionResult Update(int id, string name)
{
 if (id > 0)
 {
 _logger.LogInformation("Item has been added");
 return Content("Item added");
 }
 else
 {
 _logger.LogWarning("Invalid ID input");
 return Content("Invalid ID!");
 }
}
```

### Log Event ID

An additional measure that can enable tracking specific errors is using log event ID, which is an optional parameter that you can add to as part of a call to a log method in order to track specific event cases. Commonly, these event IDs are declared as constants and used to further distinguish the events.

By supplying an integer as the first parameter to the logging method, you can assign that event to use the specified event ID. This can help find specific events later when trying to find issues.

The following code is an example of using event ID in logs:

### Log Event ID

```
const int CALL_INDEX = 1000;

public IActionResult Index()
{
 _logger.LogInformation(CALL_INDEX, "Adding an entry to the logger.");
 return Content("Result from controller");
}
```

 **Best Practice:** A best practice is to store event IDs as enums or constants. Doing so allows you to quickly switch the ID if required. You should not provide hard coded IDs as it is difficult to maintain and understand the significance behind it.

### Logging Exceptions

Another useful parameter that can be accepted by the various logging method is exceptions. A log can be used to directly display exceptions. This is useful for handling exceptions that occur in **try/catch** blocks because later you will be able to see the exception details in the log. This can be very useful for finding issues that occur in production. The parameter for the exception should be the first parameter if event ID is omitted, or It will be the second parameter if event ID is used.

The following code is an example of exception logging:

### Exception Logging

```
public IActionResult Index()
{
 try
 {
 int x = 3;
 x -= 3;
 int result = 30 / x;
 }
 catch (Exception ex)
 {
 _logger.LogError(INVALID_DIVISION, ex, "An error occurred while dividing!");
 }
 return Content("Result from controller");
}
```

### Configure Logging

You can also utilize the **appsettings.json** file to set various details for your logging. It is most commonly used to set log levels. This can easily allow you to set a different log level in different environments. To load the configuration for logging, you will need to call the **AddConfiguration** method on the **ILoggingBuilder** parameter and supply it with a configuration by using the **GetSection** method on the **Configuration** property of the **WebHostBuilderContext** parameter. After this is done, the logging configuration will be loaded.

The following code is an example of loading logging configuration:

### Loading Logging Configuration

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
WebHost.CreateDefaultBuilder(args)
 .ConfigureAppConfiguration((hostingContext, config) =>
 {
 var env = hostingContext.HostingEnvironment;
 config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
 })
 .ConfigureLogging((hostingContext, logging) =>
 {
 logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
 if (hostingContext.HostingEnvironment.IsDevelopment())
 {
 logging.AddConsole();
 }
 else
 {
 ILoggerProvider provider = new CustomProvider();
 logging.AddProvider(provider);
 }
 })
 .UseStartup<Startup>();
```

The following code is an example of an **appsettings.json** file that contains logging configuration:

### Logging Configuration in appsettings.json

```
{
 "Logging": {
 "LogLevel": {
 "Default": "Debug"
 }
 }
}
```



**Best Practice:** Note that the configuration section is not required to be named "Logging", however, by using another name, it can cause confusion to developers working with the configuration file.

## Using Third-Party Logging Providers

In a modern application, there are innumerable ways to store logs, whether in cloud servers, databases, or other. It would be impossible to cover all possible storage methods in ASP.NET Core, and you may occasionally want to use one of many third-party logging providers. In this topic, you will learn how to easily implement Serilog file logging in your application. However, you can easily utilize other logging methods or providers such as NLog or Loggr.

The first part of importing a third-party provider is to install the relevant NuGet package. For example, if you want to use Serilog, you will need to install the **Serilog.Extensions.Logging.File** package. This will allow you to use a new provider extension named **AddFile** on the **ILoggingBuilder** parameter of the **ConfigureLogging** method. The **AddFile** method requires a file path to where the log file is saved. Simply provide a valid path, and the application will write all logs to this file.

The following code is an example of adding a third-party provider:

### Third-Party Provider

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
 WebHost.CreateDefaultBuilder(args)
 .ConfigureAppConfiguration((hostingContext, config) =>
 {
 var env = hostingContext.HostingEnvironment;
 config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
 })
 .ConfigureLogging((hostingContext, logging) =>
 {
 logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
 logging.AddFile("myLog.txt");
 })
 .UseStartup<Startup>();
```



**Note:** Note that various third-party libraries may have additional configuration steps. You should consult the provider's documentation to find out additional details. Most third-party libraries will also expose extension methods that allow you to easily add the provider. In cases where they do not, you can use the **AddProvider** method instead.

## Demonstration: How to Log an MVC Application

In this demonstration, you will see how to use logging in an ASP.NET Core application. The log messages will help to investigate problems that occur in the application.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Log an MVC Application" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD10\\_DEMO.md#demonstration-how-to-log-an-mvc-application](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD10_DEMO.md#demonstration-how-to-log-an-mvc-application).

# Lab: Testing and Troubleshooting

## Scenario

To improve the quality of a web application for a shirt store, your development team has decided to add testing and troubleshooting to the web application. You have been asked to add unit tests to test a model and a controller. You have also been told that when an error occurs the browser should display a detailed exception page on development environment, and a custom error page on the production environment. In addition, you are required to add logging to the web application.

## Objectives

After completing this lab, you will be able to:

- Test an ASP.NET Core MVC application.
- Add exception handling for the different environments.
- Add logging to an ASP.NET Core MVC application.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD10\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD10_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD10\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD10_LAK.md).

## Exercise 1: Testing a Model

### Scenario

You are required to develop an **ASP.NET Core MVC** application in a test-driven environment.

In this exercise, you will create an **MSTest** testing project and add it to the solution, add the **ASP.NET Core MVC** website application to its list of dependencies, and then test the **Shirt** model.

The main tasks for this exercise are as follows:

1. Add a testing project.
2. Write a test for a model.
3. Run the unit test – it should fail.
4. Implement the model class so the test will pass.
5. Run the unit test – it succeeds.

## Exercise 2: Testing a Controller using a Fake Repository

### Scenario

After creating a test project and testing the model, you are now required to test the controller.

To do this, you will need to create a repository interface as a dependency for the controller to gain access to the data. To test the controller, you will create a fake repository as a substitute, and then provide it to the controller via its constructor.

The main tasks for this exercise are as follows:

1. Create an interface repository.
2. Implement the interface repository using a fake repository.
3. Pass the fake repository to the constructor of a controller.
4. Write a test for a controller.
5. Run the unit test – it should fail.
6. Implement the controller class so the test will pass.
7. Run the unit test – it succeeds.

## Exercise 3: Implementing a Repository in the MVC Project

### Scenario

After the model and the controller has been tested, you have been asked to add a repository class so that the application will be able to run. In this exercise, you will implement a **ShirtRepository** repository that will get data from a database and update a database. The **ShirtRepository** repository will be registered in the **ConfigureService** method.

The main tasks for this exercise are as follows:

1. Implement the interface repository in a repository class.
2. Register the repository as a service.
3. Run the MVC application.

## Exercise 4: Adding Exception Handling

### Scenario

You have been asked to add exception handling to the web application. If an error occurs while running the application, two use cases must be implemented: In the case that the application is running in a development environment and an error occurs, the user would see a detailed error page with information on where to find the error. In the case that the application is running in a production environment, a custom none-informative page would be displayed claiming there was an error. You are required to add an exception handling to each of the use cases.

The main tasks for this exercise are as follows:

1. Add exception handling in **Startup.cs**.
2. Create a temporary exception for testing.
3. Run the application in the development environment.
4. Run the application in the production environment.
5. Remove the temporary exception.

## Exercise 5: Adding Logging

### Scenario

You are required to provide logging to the ASP.NET Core MVC application by using the **Serilog** library, while configuring the logging separately by using **appsettings.json** files to the different environments. Any trace log level logs in development would be displayed to the console, while any warning level logs in production would be written to its dedicated file. This would also require injecting the **ILogger** to the controller, thus would require to update the controller's test.

The main tasks for this exercise are as follows:

1. Add logging to the MVC application.
2. Test the controller by using a mocking framework.
3. Run the unit test.
4. Run the application in the development environment.
5. Run the application in the production environment.

**Question:** A member of your team is wondering why in the development environment the log messages are written to console, while in the production environment they are written to a file. Can you explain what is the reason for this?

**Question:** A member of your team is wondering why in development, when an exception is thrown, a developer exception page is shown. However, in production, when an exception is thrown, a custom error page is shown. Can you explain what is the reason for this?

# Module Review and Takeaways

In this module, you became familiar with various techniques that you can use to eliminate bugs from your ASP.NET Core MVC web application. This begins early in the development phase of the project when you define unit tests that ensure that each method and class in your application behaves precisely as designed. Unit tests are automatically rerun as you build the application, so you can be sure that methods and classes that did work are not broken by later coding errors. Exceptions arise in even the most well-tested applications because of unforeseen circumstances. You also saw how to ensure smooth handling of exceptions and how you can add a logging infrastructure to optionally store logs for later analysis.

## Review Question

**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

*MSTest, NUnit, xUnit.* These are unit testing frameworks. They allow setting up unit test projects for applications.

*Moq, NSubstitute.* These are mocking frameworks. They automate the creation of test doubles for unit tests.

*JSNLOG, elmah.io, Loggr, NLog, Serilog.* They are third-party providers which can grant the application more options for logging.

## Best Practices

- 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.
- 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, the time you invest will be worth it over the life of the project.
- 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
A unit test takes a long time to run or returns an error connecting to a database.	

# Module 11

## Managing Security

### Contents:

Module Overview	11-1
<b>Lesson 1: Authentication in ASP.NET Core</b>	<b>11-2</b>
<b>Lesson 2: Authorization in ASP.NET Core</b>	<b>11-18</b>
<b>Lesson 3: Defending from Attacks</b>	<b>11-28</b>
<b>Lab: Managing Security</b>	<b>11-45</b>
Module Review and Takeaways	11-47

## Module Overview

Since web applications are normally targeted towards users utilizing only a browser to use the application, there are likely to be far more users than in the case of installed applications. However, the open nature of a web application means security must always be at the forefront of your mind when building them. As part of security, you must decide which users can perform what actions, all while protecting users and your application from malicious third parties with possible access to your application.

Authentication is the act of utilizing several parameters to make sure that a user is who they claim to be. By implementing authentication, you can ascertain who a user is and provide them with appropriate content while utilizing your applications.

Authorization is the process where an already authenticated user in the application can be granted access to specific actions or resources. By utilizing authorization, you can prevent users from accessing sensitive material not intended from them or from performing actions which they should not be able to.

Finally, at some point in its lifespan, your applications may come under attack by malicious users. These can vary in means and intent, but the cost of being undefended can be great. You may lose potential users who are affected, valuable data could be erroneously changed, and in the worst cases the entire application may become unusable. Solutions to some of the most popular attacks will be reviewed in this module.

### Objectives

After completing this module, you will be able to:

- Add basic authentication to your application.
- Configure Microsoft ASP.NET Core Identity.
- Add basic authorization to your application.
- Utilize several different authorization approaches.
- Know how security exploits work and how to better defend against them.

## Lesson 1

# Authentication in ASP.NET Core

Authentication is a process by which a user can be identified within an application, and undesirable users can be prevented from accessing important data. ASP.NET Core provides you with ASP.NET Core Identity which can be used to set up authentication in your ASP.NET Core MVC application.

In this lesson, you will learn why authentication is required and how by using ASP.NET Core Identity you can add authentication to your applications. Furthermore, you will also learn about additional ways in which you can further extend your authentication, allowing you to use external login providers and external storage providers.

ASP.NET Core Identity is a powerful tool which helps you to manage both authentication and authorization, throughout your application. With ASP.NET Core you will be able to save your application users in a Microsoft SQL Server database by using tools you have learned in module 7, "Using Entity Framework Core in ASP.NET Core". After understanding the basic structure of an ASP.NET Core Identity application, you will be shown how to further configure the authentication process to protect your application from threats.

In addition, you will also learn about various ASP.NET Core providers which can be used to allow additional login options for your users, such as adding log in through external providers, as well as allowing you to further customize storage options for your authentication system.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain why authentication is needed.
- Create an ASP.NET Core MVC application utilizing ASP.NET Core Identity.
- Utilize ASP.NET Core Identity in MVC controllers.
- Configure ASP.NET Core Identity behavior.
- Describe the use of external providers with ASP.NET Core Identity.

### The Need for Authentication

When you create an application designed to provide services on a user by user basis, you need to be able to recognize individual users in your application, grant them access to information that is relevant to them, and allow them to personally interact with your application. Authentication is used to provide this functionality.

Authentication is the process of determining the identity of a user within the application. By providing a unique username alongside a password, the user can confirm their identity

within the application and you can provide specific services to the user. Identifying the user can be particularly useful in platforms for facilitating interactions such social networks or chats, websites which provide paid services such as shopping websites, as well as for websites which provide personalized

#### Why Authentication is Needed

- To identify the users that are connected to your application or website
- To ensure connected users use the correct credentials
- To enable you to block sensitive resources from unauthenticated users
- To help prevent malicious activities on your application or website

content. In addition, authentication can also be used to acquire personal data about website users. This can then be used to provide them additional services from your company.

A common type of authentication is authentication by username and password. The user will need to know both the unique username they use in the application and the related password. This is, however, not the only possible option for authentication and additional methods exist such as fingerprints, phone confirmation, email confirmation, and more.

When a single type of authentication is used, it is called single-factor authentication. For the most part it mainly uses username and password. When multiple forms of authentication are required, it is called multi-factor authentication instead and usually involves both username and password, as well as additional forms of authentication.

Creating a proper authentication environment is crucial, as it can add an important layer of security to your applications. Modern applications often involve sensitive data, and as such, require that it is only provided to appropriate users. Without authentication, data is accessible to all users, allowing malicious users to acquire sensitive information.

Most forms of authentication these days use what is known as token-based authentication. When the initial authentication is performed, a token is generated by the server and sent to the client. The client then sends the token again in every future request, allowing the server to confirm that it is the same user. If the token is incorrect or expired, the user will need to authenticate again.

## Setting up ASP.NET Core Identity

ASP.NET Core Identity is a commonly used solution for handling login functionality within ASP.NET Core applications. It is well supported within the framework and works with multiple different providers to assist developers in creating required authentication infrastructures.

As part of the ASP.NET Core Identity infrastructure Entity Framework is utilized by default in order to manage setting up and interacting with user data. In nearly all cases this will mean that you can use the same database as other parts of your application, regardless of type to handle login and logout, negating the requirement to set up additional infrastructure.

ASP.NET Core Identity also exposes various services, which you can inject throughout your application, for managing system users and performing login and logout operations. This makes adding it to your application simple and ensures that your application utilizes the various technologies you have learned previously in this course.

### Setting up the User Class

In order to set up ASP.NET Core Identity, you will first need to decide what properties you wish to store about your application users. By default, properties such as email address, username, and password hash are all stored as part of the default **IdentityUser** class which is exposed from the **Microsoft.AspNetCore.Identity** namespace.

If you require any additional properties, such as the user's first and last name, age or any other property, you will need to create your own class which inherits from the **IdentityUser** class. If you do not require any additional properties you can use the **IdentityUser** class directly instead.

ASP.NET Core Identity requires some configurations to be specified before it can be used:

- It requires an **Identity** class inheriting from **IdentityUser**
- It requires **IdentityDbContext** which is used for all database communications
- It requires a call to **AddDefaultIdentity** in **ConfigureServices**
- It requires a call to **UseAuthentication** in **Configure**

The following code is an example of inheriting from the default **IdentityUser** class:

#### Inheritance from IdentityUser

```
public class WebsiteUser : IdentityUser
{
 public string UserHandle { get; set; }
}
```

In this example, you can see the **WebsiteUser** class, which exposes the **UserHandle** property. This property could be used to determine how the user's name is displayed during online operations, without revealing the username used for login.

 **Best Practice:** Note that only a password hash should ever be stored for ASP.NET Core Identity users. Real passwords should never be stored, and it is a massive security exploit to do so.

#### Setting up the Database Context

In the next step, you will need to set up the **DbContext** class to support identity. In order to do this, the context class will need to inherit from the **IdentityDbContext** class rather than **DbContext**. You will also need to provide your chosen user identity class as a generic type to the **IdentityDbContext** class. You must provide the class you wish to use, otherwise you will not be able to access the appropriate class properties.

A nice advantage when using **IdentityDbContext** is that you will not need to add a **DbSet** instance to manage users. The logic for managing users is built in as part of **IdentityDbContext** itself.

The following code is an example of setting up an **IdentityDbContext**:

#### IdentityDbContext

```
public class AuthenticationContext : IdentityDbContext<WebsiteUser>
{
 public AuthenticationContext(DbContextOptions<AuthenticationContext> options) :
 base(options)
 {
 }
}
```

In this example, you can see that the **AuthenticationContext** class inherits from **IdentityDbContext** and provides the **WebsiteUser** custom user class instead of the default **IdentityUser**. It is otherwise identical in structure to **DbContext**.

 **Best Practice:** You can use the **IdentityDbContext** to manage additional datasets, in the same way as **DbContext**. It adds additional functionality on top of **DbContext** and isn't intended to be used exclusively for authentication.

#### Configuring Identity in Startup Class

Once the database context has been established, you will need to add calls to the ASP.NET Core Identity services and middleware.

Inside the **ConfigureServices** method, in the **Startup** class, you will need to call the **AddDbContext** method on the **IServiceCollection** parameter with a class which inherits from **IdentityDbContext**. It is instantiated in the same way as any other **DbContext**, with the same parameters.

You will also need to add a call to the **AddDefaultIdentity<\*User\*> ()** method. The **AddDefaultIdentity** method expects the class which will be used to manage users in the application. This can be either a custom class which inherits from **IdentityUser** or it can be **IdentityUser** itself.

You should also pipe a call to **AddEntityFrameworkStores<\*DbContext\*> ()** immediately after **AddIdentity**. This is used to connect the **IdentityDbContext** to the **IdentityServices** added by the call to **AddIdentity**. It expects a database context type to be provided and you should use the same context class as the one you created for managing identity.



**Note:** If you do not wish to use Entity Framework in your application you should not make a call to **AddEntityFrameworkStores**.

The following code is an example of **ConfigureServices** adding identity services:

### Identity Services

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();

 services.AddDbContext<AuthenticationContext>(options =>
 options.UseSqlite("Data Source=user.db"));

 services.AddDefaultIdentity<WebsiteUser>()
 .AddEntityFrameworkStores<AuthenticationContext>();
}
```

In this example, you can see that the services for ASP.NET Core Identity are instantiated by the call to the **AddDefaultIdentity** method. This application uses the **WebsiteUser** class to manage users. It then connects the identity services to the **IdentityDbContext** provided by **AuthenticationContext**.

Finally, in the **Configure** method of the startup class, you will need to load the middleware for handling the authentication process in order to enable Identity. You can do this by calling the **UseAuthentication** method of the **IApplicationBuilder** object. It is important that you call **UseAuthentication** before the call to **UseMvc**. This is to ensure that the authentication will always be ready for use within our ASP.NET Core MVC Controllers.

The following code is an example of enabling identity middleware:

### Identity Middleware

```
public void Configure(IApplicationBuilder app, AuthenticationContext authContext)
{
 authContext.Database.EnsureDeleted();
 authContext.Database.EnsureCreated();

 app.UseAuthentication();
 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("");
 });
}
```

## Interfacing with ASP.NET Core Identity

Once authentication has been set up inside of your ASP.NET Core MVC application, you are able to start using it throughout your application.

### Performing Login

One of the most common flows in the authentication process is logging in. Logging in is the process of the user entering their credentials in a dedicated form in an application and the server confirming that the credentials are valid before sending an authentication token to the client, which will be used to check the user identity on future requests. To perform this initial connection, you will need to be able to handle a login request on the server.

To facilitate login, you should create a controller named **AccountController**. While you can use a different name for it, much of the default logic in ASP.NET Core MVC relies on **AccountController** and using a separate name will require additional work.

In order to be able to perform login operations, you will need to inject the **SignInManager<T>** service, where **T** is the class which inherits from **IdentityUser** as described in the previous topic. The **SignInManager** class exposes valuable methods for performing authentication operations. For example, to login you will need to call the **PasswordSignInAsync(\*username\*, \*password\*, \*isPersistant\*, \*lockoutOnFailure\*)** method.

The **PasswordSignInAsync** method is designed to authenticate a user and create a session, while also updating the client with the authentication token required for future requests. It accepts parameters as follows:

**\*username\***. The username used by the user. String variable.

**\*password\***. The users non-hashed password, the non-hashed password should never be stored anywhere as that is a security exploit. String variable.

**\*isPersistant\***. Determines if the cookie created by the sign in method persists after the browser is closed. Boolean variable.

**\*lockoutOnFailure\***. Determines if the user will be locked out if the login attempt is unsuccessful. Lockout can be used to prevent malicious attempts to login at the cost of inconveniencing the user if the wrong password is typed. Boolean variable.

Once the authentication method succeeds, you can then redirect the user to an appropriate URL. In particular, a handy feature of ASP.NET Core Authorization, which will be covered in further detail in the next topic, "Authorization in ASP.NET Core", is the addition of a **RedirectUrl** parameter in the query string. By using this parameter, you are able to navigate to the original destination page which requires authentication when it is successful.

You will also need to create a view which will call the login method on the controller. Note that the view should use a different model than the one used by the **DatabaseContext**. Many properties in the **UserIdentity** are not likely to be used, while other, more sensitive properties such as passwords should be present in the model used by the view, while they should be stored in a hashed format on the class derived from **UserIdentity**.

ASP.NET Core MVC utilizes two main services for authentication

- **SignInManager** – Service designed to perform authentication operations. This includes logging in and out of the application.
- **UserManager** – Service designed to perform operations on users themselves. Can be used to add or remove, as well as find users.

In addition, the **Controller User** property exposes **Identity** which allows us to get information about the current user.



**Best Practice:** You may sometimes find that you require separate models for server data and client data. A common example for this is when using password. While the user will need to input his password on the client, it should not be stored within the database. As such, you will need a client-side model which does store a password, and a server-side model which does not retain the password. To differentiate this, you can name models which are intended to be used on the client side with **ViewModel** at the end. This will make them easily visible as models intended for usage within the view.

The following code is an example of a ViewModel which can be used for login:

### Login ViewModel

```
public class LoginViewModel
{
 public string Username { get; set; }
 public string Password { get; set; }
}
```

Note that the password in this view model will not be hashed and should not be saved.

The following code is an example of a view for login:

### Login View

```
@model Authentication.Models.LoginViewModel

<h2>Login</h2>
<div>
 <form method="post" asp-action="Login">
 <div>
 <label asp-for="Username">Username</label>
 <input asp-for="Username" />
 </div>
 <div>
 <label asp-for="Password">Password</label>
 <input asp-for="Password" />
 </div>
 <input type="submit" value="Login" />
 </form>
</div>
```

The following code is an example of an **AccountController** which can handle Login:

### Login Method

```
public class AccountController : Controller
{
 private readonly SignInManager<WebsiteUser> _signInManager;

 public AccountController(SignInManager<WebsiteUser> signInManager)
 {
 _signInManager = signInManager;
 }

 [HttpPost]
 public async Task<IActionResult> Login(LoginViewModel model)
 {
 if (ModelState.IsValid)
 {
 var result = await _signInManager.PasswordSignInAsync(model.Username,
model.Password, true, false);
 if (result.Succeeded)
 {
 if (Request.Query.Keys.Contains("ReturnUrl"))
```

```
 {
 return Redirect(Request.Query["ReturnUrl"].First());
 }
 else
 {
 return RedirectToAction("Index", "Home");
 }
 }
}
return View();
}
```

As you can see in this example, there is an **AccountController** class which receives a **SignInManager** instance with the **WebsiteUser** type through dependency injection. Whenever a POST request is made for the **Login** method, the model state is validated, and if it is valid, the username and password are used to perform sign in. If the sign in is successful, it will check if a **ReturnUrl** is available. If it exists, it will redirect to the originally intended page. Otherwise, it will navigate to the home page. In all other cases, it will reload the login view.

 **Note:** Note that since **PasswordSignInAsync** is an **async** method, you will need to wrap the method in a **Task**, and use await to determine if the login is successful.

## Performing Logout

Occasionally, users that are logged in to your application will wish to logout of the application. This can be for a variety of reasons, such as switching to another user or preventing access to their account from potentially malicious sources nearby. To facilitate this and to allow users to log out at their discretion, you will want to add logic for handling logout requests.

Similarly, to login, logout should also be on the same controller, although it is possible to separate them if it makes more sense for your application. It also utilizes the **SignInManager** service to perform the logout.

An additional step you should take before performing a logout is to check if there is a currently authenticated user logged in to the current session. This can be done by using the **User** property of the **Controller** class. The **User** property exposes various properties relating to the current user in the session. To check if a user has been successfully authenticated, you can check the **IsAuthenticated** property under the **Identity** property in the **User**. This will return the current authentication status and you can use it to make decisions on how to handle the user throughout your application.

To actually perform a logout, you will need to call the **SignOutAsync()** method from the **SignInManager**. This method clears the stored identity tokens for the user, ensuring that future calls will not be on the current user, unless a new login is performed.

The following code is an example of a logout view:

### Logout View

```
Log out
```

The following code is an example of a logout method on the controller:

### Logout method

```
public IActionResult Logout()
{
 if (User.Identity.IsAuthenticated)
 {
 _signInManager.SignOutAsync();
 }
 return RedirectToAction("Index", "Home");
}
```

Note that in this example a check is made to see if the current user is authenticated before performing the sign out. This is done by checking the **User.Identity.IsAuthenticated**.

### Adding New Users

Another common occurrence in web applications is adding new users to the database. By adding the option to register new users in your web application, you open your application to be able to serve many different users and not have to rely on a manually maintained list.

Unlike login and logout, adding new users does not depend on the **SignInManager** service. Instead you will need to inject the **UserManager<T>** service, with the **T** being the type that inherits from **IdentityUser**, which is used for modifying existing users. You can use the **CreateAsync(\*user\*, \*password\*)** method to create a new user in the system.

The **CreateAsync** method will create a new user in the database by using the various user properties stored on your chosen class, while hashing the password into a more secure hash string. This helps ensure that the actual password is not stored anywhere, thereby maintaining the application security.

Once the user has been created successfully, the user will be able to log in with the new credentials and you can then use the user details to login the user.

The following code is an example of a view model for registering new users:

### Register ViewModel

```
public class RegisterViewModel : LoginViewModel
{
 public string UserHandle { get; set; }
}
```

Note that since the registration process also requires the **Username** and **Password** properties, it is logical to extend the **LoginViewModel** class.

The following code is an example of a view for registering new users:

### Registration View

```
@model Authentication.Models.RegisterViewModel

<form method="post" asp-action="Register">
 <div>
 <label asp-for="Username">Username</label>
 <input asp-for="Username" />
 </div>
 <div>
 <label asp-for="Password">Password</label>
 <input asp-for="Password" />
 </div>
 <div>
 <label asp-for="UserHandle">Display Name</label>
 <input asp-for="UserHandle" />
 </div>
 </form>


```

```

 </div>
 <input type="submit" value="Register" />
 </form>
</div>

```

The following code is an example of a registration method on a controller:

### Register Method

```

public class AccountController : Controller
{
 private readonly SignInManager<WebsiteUser> _signInManager;
 private readonly UserManager<WebsiteUser> _userManager;

 public AccountController(SignInManager<WebsiteUser> signInManager,
 UserManager<WebsiteUser> userManager)
 {
 _signInManager = signInManager;
 _userManager = userManager;
 }

 [HttpPost]
 public async Task<IActionResult> Register(RegisterViewModel model)
 {
 if (ModelState.IsValid)
 {
 WebsiteUser user = new WebsiteUser
 {
 UserHandle = model.UserHandle,
 UserName = model.Username,
 };

 var result = await _userManager.CreateAsync(user, model.Password);
 if (result.Succeeded)
 {
 return await Login(model);
 }
 }
 return View();
 }
}

```

In this example, you can see that both **SignInManager** and **UserManager** are injected in the constructor. When performing the **Register** action, you can see that once the **ModelState** is validated, a new **WebsiteUser** is created and a password property is not filled in. Afterward, the **CreateAsync** method of the **UserManager** is called with the new user and password, which will be hashed. Finally, if the user is created successfully, the **Login** method will be called.

### Accessing the User Properties

Finally, once the log in flow has been completed, the last step is accessing the properties of your user. This can be done by using the **User** property alongside the **UserManager** service.

By using the **FindByNameAsync(\*name\*)** method on the **UserManager** object and providing it with the value of the **Name** property on **User.Identity**, the **UserManager** will retrieve the user, if it is found.

Afterward, you are able to extract properties off the user class you selected and apply them to the page model as is needed.

The following code is an example of a view model for displaying the user:

### User Display ViewModel

```
public class UserDisplayViewModel
{
 public string UserHandle { get; set; }
 public string UserName { get; set; }
}
```

The following code is an example of a view which displays custom **IdentityUser** properties:

### Displaying IdentityUser Properties

```
@model IdentityAgain.Models.UserDisplayViewModel

@if (Model != null)
{
 <div>Hello @Model.UserHandle</div>
}
```

The following code is an example of a controller retrieving the properties for **IdentityUser**:

### Getting IdentityUser Display Properties

```
public async Task<IActionResult> Index()
{
 if (!User.Identity.IsAuthenticated)
 {
 return RedirectToAction("Login", "Account");
 }
 WebsiteUser user = await _userManager.FindByNameAsync(User.Identity.Name);
 if (user != null)
 {
 UserDisplayViewModel model = new UserDisplayViewModel
 {
 UserHandle = user.UserHandle,
 UserName = user.UserName
 };
 return View(model);
 }
 return View();
}
```

In this example, you can see that this action will only display for authenticated users. Furthermore, it then attempts to retrieve the user for the currently logged in user with the **WebsiteUser** custom class which inherits from **IdentityUser**. It then uses it to create a new view containing properties from the **WebsiteUser** class, including **UserHandle** which is not present on the default **UserIdentity** class.

## ASP.NET Core Identity Configuration

Until now you have learned how to set up basic end to end configuration for ASP.NET Core Identity. However, so far, the applications you have created all utilize the default settings. In a real application, you have to set up requirements such as password settings, user configurations, session settings, and more.

Many of these settings can be set up in the **ConfigureServices** method as part of the **AddDefaultIdentity** method. It is able to optionally receive an **IdentityOptions** parameter which can be used to configure a variety of additional settings.

In this lesson, you will learn several ways in which the **IdentityOptions** can be used to further customize the behavior for ASP.NET Core Identity

The following code is an example of calling **AddDefaultIdentity** with **IdentityOptions**:

### Configuring IdentityOptions

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();

 services.AddDbContext<AuthenticationContext>(options =>
 options.UseSqlite("Data Source=user.db"));

 services.AddDefaultIdentity<WebsiteUser>(options =>
 {
 /* Configure Identity Options options here */
 })
 .AddEntityFrameworkStores<AuthenticationContext>();
}
```

There are many things you can configure in ASP.NET Core Identity configuration:

- User settings – Allow to configure what constitutes a legal username and other options
- Lockout settings – Allow to customize lockout behavior if incorrect passwords are supplied
- Password settings – Allow setting password complexity rules
- Sign in settings – Allow requiring additional methods of authentication before creating users
- Cookie settings – Allow changing the behavior of cookies on the website

### User Settings

User settings are used to determine a variety of configurations regarding the registration of users in the system. Settings here can be used to require a unique email address for every configured user or to determine which letters and symbols can be used in the application.

You can use the **User** property of **IdentityOptions** to configure settings relating to the **IdentityUser**.

The following code is an example of configuring settings related to the identity user:

### User Settings

```
services.AddDefaultIdentity<WebsiteUser>(options =>
{
 options.User.RequireUniqueEmail = true;
 options.User.AllowedUserNameCharacters =
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 -._@+";
})
 .AddEntityFrameworkStores<AuthenticationContext>();
```

In this example, you can see that all users in the application will require a unique value for the **Email** property to be set and that only characters from the following:

**"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 -.\_@+"** can be used in the **Username** property.

## Lockout Settings

An important feature which is frequently present in various systems involving authentication is the concept of user lockout. When the application detects multiple attempts to log in with the same username but invalid passwords, it can be set to lock the user for future logins for a period of time. This can prevent malicious sources from logging in to the system by utilizing brute force tactics which involve attempting to guess a password, by vastly increasing the time between attempts rendering them unviable.

To apply the lockout, you will need to make sure the **lockoutOnFailure** parameter in the call to the **PasswordSignInAsync** method is true, or the lockout settings will be ignored.

You can use the **Lockout** property of the **IdentityOptions** to configure settings relating to the lockout.

The following code is an example of enabling lockout on login:

### Enabling Lockout

```
var result = await _signInManager.PasswordSignInAsync(model.Username, model.Password,
true, true);
```

The following code is an example of lockout settings in Identity:

### Lockout Settings

```
services.AddDefaultIdentity<WebsiteUser>(options =>
{
 options.Lockout.AllowedForNewUsers = true;
 options.Lockout.MaxFailedAccessAttempts = 3;
 options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(1);
})
 .AddEntityFrameworkStores<AuthenticationContext>();
```

In this example, you can see that lockout can occur for new users who have not yet logged in successfully. The users will be locked out for one minute and users can attempt to log in up to three times before being locked out.

## Password settings

Secure passwords are a requirement in modern authentication environments and it is the developer's duty to assign a password requirement that is appropriate for the specific environment. While in many cases a default password can be used, sometimes you will find that you need to customize it further.

In order to configure password settings and complexity, you can access the **Password** property of the **IdentityOptions**.

The following code is an example of password configuration in Identity:

### Password Settings

```
services.AddDefaultIdentity<WebsiteUser>(options =>
{
 options.Password.RequiredLength = 10;
 options.Password.RequiredUniqueChars = 3;
 options.Password.RequireDigit = true;
 options.Password.RequireNonAlphanumeric = true;
 options.Password.RequireUppercase = true;
 options.Password.RequireLowercase = false;
})
 .AddEntityFrameworkStores<AuthenticationContext>();
```

In this example, you can see that passwords for this application will be at least 10 characters long, require at least three different characters to be used, require a numeric character, as well as at least one symbol character. Finally, at least one uppercase character will be required. Lower case characters will not be required and can be omitted.

## Sign In

In modern applications, you may sometimes require that users provide additional details that allow you to access them for identity verification. This can be helpful in notifying them of potential attempts to compromise their accounts.

The **SignIn** property of the **IdentityOptions** can be used to require the user to confirm their email or phone number in the application. Login will be impossible until successful confirmation occurs.

 **Note:** Setting up both email and mobile number confirmation requires additional work and will not be covered in this course.

The following code is an example of the **SignIn** configuration in Identity.

### SignIn Settings

```
services.AddDefaultIdentity<WebsiteUser>(options =>
{
 options.SignIn.RequireConfirmedEmail = true;
 options.SignIn.RequireConfirmedPhoneNumber = false;
})
 .AddEntityFrameworkStores<AuthenticationContext>();
```

In this example, you can see that the application requires users to confirm registration via email. Confirming phone numbers is not required.

## Cookies Settings

An additional type of setting that affects Identity is the cookie configuration. By calling the **ConfigureApplicationCookie** method in **ConfigureServices**, you can use a **CookieAuthenticationOptions** parameter to set various properties for cookies in your application. Various settings such as cookie name and expiration time can be set here and will apply to the cookies used by the application.

 **Note:** Note that this setting affects application cookies as a whole rather than just Identity. When using it, you should plan accordingly.

The following code is an example of cookie configuration:

### Cookies Settings

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();

 services.AddDbContext<AuthenticationContext>(options =>
 options.UseSqlite("Data Source=user.db"));

 services.AddDefaultIdentity<WebsiteUser>(options =>
 {
 })
 .AddEntityFrameworkStores<AuthenticationContext>();

 services.ConfigureApplicationCookie(options =>
```

```

 {
 options.Cookie.Name = "AuthenticationCookie";
 options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
 options.SlidingExpiration = false;
 });
}

```

In this example, you can see that cookies are configured separately from Identity and are not part of it. The cookies are configured to use the name "**AuthenticationCookie**" and they will last up to 30 minutes from the time of creation. They will not be refreshed.

## Demonstration: How to use ASP.NET Core Identity

In this demonstration, you will learn how to configure ASP.NET Core Identity in startup class, and how to use **SignInManager** and **UserManager** in the **AccountController** class.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to use ASP.NET Core Identity" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD11\\_DEMO.md#demonstration-how-to-use-ASP.NET-core-identity](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD11_DEMO.md#demonstration-how-to-use-ASP.NET-core-identity)

## Customizing Providers with ASP.NET Core

Out of the box ASP.NET Core Identity utilizes Entity Framework to handle authentication and handles the end to end authentication within the application scope. This can be fine in many cases, but occasionally you may find yourself needing to utilize additional options. Business requirements may dictate a need to use the internal active directory infrastructure to login or you may wish to allow users of alternate frameworks to use your application, or even possibly be unable to use SQL Server for data storage.

- By default, ASP.NET Core Identity uses Entity Framework and the built-in authentication logic
- You can extend Identity to allow authentication from multiple external sources
- You are also able to use Windows Active Directory to handle authentication for you, to prevent external users from accessing your application
- Furthermore, the backend can be fully customized to use any form of storage you require, as long as it can be used in an ASP.NET Core application

In this chapter, you will be introduced to several potential customization options for your applications. Note that all of these require extensive changes to your application and should be worked on and decided in advance.

### Windows Authentication

When creating an application designed to be used in an intranet environment, you should consider using Windows Authentication in your application. By setting up Windows Authentication, users will be able to sign into the application by using their Windows credentials.

By using Windows authentication, the active directory will be used to determine user credentials and permissions, allowing users to be managed and created by internal administrators. This helps create safer applications and removes the need to manually manage users.

You may wish to use Windows Authentication when:

- Your application is designed to be used within an organization.
- When you want user creation to be limited and managed manually.
- When you want to prevent undesirable users in your application.
- When your applications feature information sensitive to your organization.

You should not use Windows Authentication when:

- When you need to create applications which can be accessible to the wider internet.
- When you want users to be able to register and create their own accounts in your application.
- When your organization is not managed by Windows Active Directory

## Utilizing External Providers

By default, in an ASP.NET Core application all logins are handled through your application itself. However, the registration process can often be tedious and many users already possess multiple different users across many different websites.

To help simplify this, you can implement signing in by using various external providers. Examples of external providers include:

- Microsoft – As used by a variety of Microsoft products, such as Microsoft Azure or Hotmail.
- Facebook – As used on Facebook and its mobile application.
- Twitter – As used on Twitter and its mobile application.
- Google – As used for Google applications such as Gmail and Google Maps.
- And many more.

By implementing support for an external provider, you allow your users to sign in through a service in which they already have an account, reducing the amount of bookkeeping they require for their accounts, and enticing users who do not wish to run through complex registration procedures.

When external providers are used, the user is still saved in your storage; however, many of the user details remain with the provider. As a result of a successful connection, your application will receive a success token which can then be used to get additional information about the user.

 **Note:** Due to security concerns, most providers will notify the users about which details become accessible to your application during the first login. This is done by the various third-party providers to ensure personal information is not passed without the user's consent. You can configure your application to request additional details and the user will be notified accordingly.

There is no single uniform way of configuring providers, as each provider utilizes different protocols and logic. As a result, you should always consider the cost/benefit of adding each provider rather than providing as many as you can.

 **Note:** External providers can be used alongside the default ones and it may even be wise to offer the option to users without accounts

## Using Alternative Storage Providers

By default, ASP.NET Core MVC utilizes Entity Framework Core to manage users in the application. This can be convenient for setting up simple storage solutions for your application. However, it is not always a viable option.

Many times during the development process, you may find that you are required to utilize a different storage option and not be able to use SQL Server or even utilize data in a different format. Examples of possible scenarios include but are not limited to:

- Azure Table Storage.
- Databases with a different data structure such as MongoDB or Redis.
- Utilizing Data Access systems other than Entity Framework Core.

In order to support these storage providers, you will need to write a new custom provider to support your chosen storage method. The specific code depends on the chosen provider. To implement it, you will need to create alternative identity store logic, as well as alternative data access layer.

The store is used to set up the datatypes you use for the application. These may be the default types as implemented in ASP.NET Core; however, they will need to be created to work with your chosen dataset. The store determines the setup of the basic entities and will need to include properties such as an **IdentityUser** derived class, as well as details about the login process related to external providers.

Meanwhile, the data access layer is used for providing the logic for communication with your chosen method of storage. It will need to be created to work with the chosen storage mechanism, but is easily customizable and extensible. When utilizing a storage provider most of your work is likely to occur in the database access layer.

## Lesson 2

# Authorization in ASP.NET Core

Authorization is the process of determining what a user is allowed to do in an application. For example, in a library application, an administrative user may approve checking out books, adding books, loaning books, and throwing away destroyed books. While a regular user in the library can make requests to lend books or browse them online.

In this lesson, you will learn why authorization is a crucial part of many applications and how you can utilize it to restrict access within your application. Furthermore, you will learn about several of the options available to you to further refine the authorization process, allowing you to set the requirement criteria you need.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the need for authorization.
- Explain why there is a need to restrict access to resources.
- Using the simple authorization attribute.
- Describe different authorization methods.
- Implement different authorization methods.

### Introduction to Authorization

In many applications, you will want to prevent certain users from accessing various resources in your application. Whether you wish to prevent users from accessing specific pages, specific data, or specific actions, this can be handled by the ASP.NET Core Authorization infrastructure.

 **Best Practice:** While it is possible to perform authorization logic by entirely relying on the **SignInManager** and its various functions, the built-in authentication systems are much simpler and offer a robust number of options.

- The **AuthorizeAttribute** attribute:
  - Restricts user access to information
  - Mandates that users should be authorized to access the authorized controller or action
- The **AllowAnonymousAttribute** attribute:
  - Allows users access to an action in a controller with the **Authorize** attribute

ASP.NET Core authorization lets you create and implement different kinds of authorization within your application and utilize it to limit content as is appropriate. For example, you can block your application's store page by using simple authorization, you can prevent adult products displaying for younger users by setting claims authorization, or you can allow only administrators to access financial details about your website by using role authorization.

## Setting Up Authorization

Authentication is also relatively simple to set up, requiring only a small change in the call to **ConfigureServices**. Instead of the call to the **AddDefaultIdentity<\*User\*>()** method on the **IServicesCollection** parameter, you will need to instead call the **AddIdentity<\*User\*, \*Role\*>()** method. The **User** parameter should remain the same, however the **Role** parameter is the **IdentityRole** class or a class derived from **IdentityRole**.

The **IdentityRole** class is designed to manage roles within the application, similarly to how **IdentityUser** manages users and it is used to start up the **RoleManager** class. However, unlike **IdentityUser**, which is often extended, **IdentityRole** isn't commonly extended and the base class is most frequently used.

While the names of **AddDefaultIdentity** and **AddIdentity** are similar, they operate very differently and if you wish to use ASP.NET Core MVC logic alongside authorization, you will need to utilize **AddIdentity**. If you try and use **AddDefaultIdentity** in an ASP.NET Core MVC application, you may find that redirection paths from failed authorization attempts will cause you to navigate to invalid paths to the **AccountController**. This is due to **AddDefaultIdentity** utilizing an ASP.NET Core Razor Pages infrastructure.

The following code is an example of calling **AddIdentity**:

### AddIdentity

```
services.AddIdentity<WebsiteUser, IdentityRole>()
 .AddEntityFrameworkStores<AuthenticationContext>();
```

 **Note:** If you wish to continue utilizing **AddDefaultIdentity** in your applications, you can pipe a call to **AddRoles<\*Role\*>()** after the call to **AddDefaultIdentity**. Note that this will require you to manage authentication by using ASP.NET Core Razor Pages.

## Simple Authorization

The most basic form of authorization revolves around blocking users who are not signed in from accessing certain pages or performing specific actions. This is called simple authorization, and it automatically redirects to the **Account\Login** action, alongside a return URL for easy reconnection. This allows you to easily gate off entire controllers or individual actions and prevents you from requiring to manually configure redirections.

In order to implement simple authorization, all you require is to add the **[Authorize]** attribute. This attribute can be added to a controller class, preventing unauthorized users from accessing the class at all while not logged in. Users who are not signed in but attempt to navigate to the controller will be redirected to the Login URL.

The **Authorize** attribute isn't limited to the controller class and can also be added to individual actions. This can be valuable for example when setting up a controller you wish to be publicly accessible but with specific actions blocked from users who have not yet logged in. For example, this could include browsing a shop being publicly accessible while actually making a purchase requiring logging in.

Alternatively, you may wish to limit most actions of a controller, while allowing one or two exceptions. While you can theoretically add **Authorize** to every action manually, it could result in an action accidentally being added without authorization to a secure controller, resulting in a security breach. Instead, in such a case, it would be better to add **Authorize** to the controller itself and add the **[AllowAnonymous]** attribute to actions which are exceptions. By setting **AllowAnonymous**, you enable access to the action itself while the rest of the controller is not accessible. An example for this could be a web page that allows you to view data, while requiring sign in to actually make changes.

The following code is an example of simple authorization on a controller:

### Simple Authorization on a Controller

```
[Authorize]
public class AuthorizedController : Controller
{
 private readonly UserManager<WebsiteUser> _userManager;

 public AuthorizedController(UserManager<WebsiteUser> userManager)
 {
 _userManager = userManager;
 }

 public async Task<IActionResult> Index()
 {
 WebsiteUser user = await _userManager.FindByNameAsync(User.Identity.Name);
 if (user != null)
 {
 UserDisplayViewModel model = new UserDisplayViewModel
 {
 UserHandle = user.UserHandle,
 UserName = user.UserName
 };
 return View(model);
 }
 return View();
 }
}
```

In this example, the authorization requirement is on the controller itself. All actions in the controller are blocked unless a user is authenticated.

The following code is an example of simple authorization on an action:

### Simple Authorization on an Action

```
public class ProductController : Controller
{
 private IShop _shop;

 public ProductController(IShop shop)
 {
 _shop = shop;
 }

 public IActionResult Index()
 {
 return View(_shop.GetAllProducts());
 }

 public IActionResult Get(int productId)
 {
 return View(_shop.GetProduct(productId));
 }

 [Authorize]
 public IActionResult BuyProduct(int productId)
 {
 _shop.PurchaseProduct(productId);
 return View();
 }
}
```

In this example, only the **BuyProduct** action requires authentication. Users can freely view products in the store, but they need to be authenticated to buy a product.

The following code is an example of **AllowAnonymous** in a controller with authorization:

### AllowAnonymous

```
[Authorize]
public class RecipeController : Controller
{
 private ICookbook _cookbook;

 public RecipeController(ICookbook cookbook)
 {
 _cookbook = cookbook;
 }

 [AllowAnonymous]
 public IActionResult Index()
 {
 return View(_cookbook.GetAllRecipes());
 }

 public IActionResult Get(int productId)
 {
 return View(_cookbook.GetRecipe(productId));
 }

 public IActionResult AddRecipe(Recipe recipe)
 {
 _cookbook.AddRecipe(recipe);
 return View();
 }
}
```

In this example, you can see that a cookbook controller is created. All users will be able to get the list of recipes, but they will not be able to access the pages for individual recipes. Only authenticated users will be able to see individual recipes or add new ones.

## Authorization Options

As you have seen the most basic form of authorization involves preventing access to users who are not logged in. However, most of the time, your requirements may be more nuanced than that. Perhaps you wish to only allow administrators access to certain pages, or require specific user details.

In this topic, you will learn of additional commonly used options for authorization inside of your web applications. These include, but are not limited to, role-based authorization, claims-based authorization and policy-based authorization.

There are many different ways to block users from accessing resources.

- You may want certain resources or actions only available to administrative users and can implement this by using roles
- You might want users to provide specific information before allowing them access to certain resources and can implement this by using claims
- You might want a completely unique logic for directing access and can implement this by using custom policies

### Role-Based Authorization

A simple and common form of authorization, role-based authorization determines what users can access based on their roles within the system. There are no default roles defined within ASP.NET Core applications, however you can create your own roles to fit the requirements of your applications.

The first step towards working with roles is by populating the roles within the system. This can be done by using the **RoleManager<\*Role\*>** service which is instantiated by the call to **AddIdentity** within the **ConfigureServices** method. The **RoleManager** will be of the same type provided for the role in the call to **AddIdentity**, usually **IdentityRole**, and can be injected throughout the application allowing you to manage roles.

## Creating Roles

In order to create a role, you will need to call the **CreateAsync(\*role\*)** method on the role manager. The **role** parameter expects a role of the same type which is configured by the **AddIdentity** method. To instantiate the basic role, all you require is to provide a role name.

You can, at any point, check if a given role exists by calling the **RoleExistsAsync(\*role name\*)** method. It accepts a string name, and checks if this specific role has already been created.

 **Note:** Generally, you can create roles at any point throughout the application, but it is recommended that you create them as part of the database instantiation, or in the **Configure** method. If you intend to create them in the **Configure** method, ensure that the database exists before creating any roles.

The following code is an example of a method in the startup class for setting up roles:

## Creating Roles

```
public async void CreateRoles(RoleManager<IdentityRole> roleManager)
{
 string[] roleNames = { "Administrator", "Manager", "User" };
 foreach (var roleName in roleNames)
 {
 bool roleExists = await roleManager.RoleExistsAsync(roleName);
 if (!roleExists)
 {
 IdentityRole role = new IdentityRole();
 role.Name = roleName;
 await roleManager.CreateAsync(role);
 }
 }
}
```

In this example, you can see that three roles are created in the application. In addition, before creating each role, if a role already exists, for example if the database was created at a previous time, it will not attempt to recreate the role. This can also allow you to add additional roles in the future. The **"Administrator"**, **"Manager"** and **"User"** roles will all be created.

## Assigning Roles to Users

In order to give a user a role, you will need to assign one or more roles to the user. This can be done at the time of registration or through different processes. For example, you may wish to give all users that register a **"User"** role, but only grant them administrator privileges from another admin.

In order to grant a user a role, you will need to call the **AddToRoleAsync(\*user\*, \*role name\*)** method. This method expects a user of the type defined by the user manager, as well as a role name string which is used to connect the user to an existing role. If the role has not been created, then the user will not be assigned to it.

The following code is an example of assigning roles to a user:

### Assigning Roles

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
 if (ModelState.IsValid)
 {
 WebsiteUser user = new WebsiteUser
 {
 UserHandle = model.UserHandle,
 UserName = model.Username,
 Email = model.Email
 };

 var result = await _userManager.CreateAsync(user, model.Password);

 if (result.Succeeded)
 {
 var addedUser = await _userManager.FindByNameAsync(model.Username);

 await _userManager.AddToRoleAsync(addedUser, "Administrator");
 await _userManager.AddToRoleAsync(addedUser, "User");

 return await Login(model);
 }
 }
 return View();
}
```

In this example, you can see that a new user is created and assigned to both the "**User**" and "**Administrator**" roles.

### Authorizing with Roles

Finally, to authorize with a role, you can use the **Authorize** attribute. By providing a **Role** parameter to the attribute, you can determine which role or roles can access the class or method decorated by the attribute. This parameter is always a string and should match the name of one of the roles that were created. If more than one role can access the class or method, you can separate all valid role values by a comma.

The following code is an example of role authorization:

#### Role Authorization

```
[Authorize(Roles = "Administrator")]
public async Task<IActionResult> DeleteUser(string username)
{
 var user = await _userManager.FindByNameAsync(username);
 if (user != null)
 {
 await _userManager.DeleteAsync(user);
 }
 return View();
}
```

In this case, only "**Administrator**" users will be able to delete users in the application.

The following code is an example of the syntax for giving access to multiple roles:

### Granting access to multiple roles.

```
[Authorize(Roles = "Administrator, Manager")]
```

## Claims-Based Authorization

Another common method of authorization in ASP.NET Core is the system of claims. A claim is a key-value pair which defines the user itself, rather than the user's permissions. Examples of claims include email or address and it is even possible to add custom claims.

Claims can be declared as is in an ASP.NET Core MVC application by just using a key and a value, but you can also optionally choose to add **Issuer** to claims. This denotes where the claim comes from and can be used to identify the source of the data. This can be useful when you have an application which communicates with other applications or providers and want to verify that the data is certified by a trusted source.

Claims-based authorization is used to determine if the user has a specific type of claim or not. This can be useful for whenever you require the user to input specific details before accessing specific pages. For example, you may require that the user provide an email address in order to allow the user to register for a newsletter or you may require that the user input a street address to allow access to a shipping button.

### Creating a Claim

To create a new claim, you can use the **Claim** class. The simplest constructor, **Claim(\*claim type\*, \*claim value\*)**, receives a claim type, which is a string denoting the key for the claim and a value which is the value for the claim. You can also utilize the **ClaimTypes** enum as the first parameter to provide a large variety of common claims.

After you have created a claim, you can use the **AddClaimAsync(\*user\*, \*claim\*)** method on the **UserManager** to add the claim to a specific user. The user being of the same type as the one used by the **UserManager** and the claim being the newly created **Claim** object.

The following code is an example of adding a claim to a new user:

### Adding a Claim

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
 if (ModelState.IsValid)
 {
 WebsiteUser user = new WebsiteUser
 {
 UserHandle = model.UserHandle,
 UserName = model.Username,
 Email = model.Email
 };

 var result = await _userManager.CreateAsync(user, model.Password);

 if (result.Succeeded)
 {
 var addedUser = await _userManager.FindByNameAsync(model.Username);

 if (!string.IsNullOrWhiteSpace(model.Email))
 {
 Claim claim = new Claim(ClaimTypes.Email, model.Email);
 await _userManager.AddClaimAsync(addedUser, claim);
 }

 return await Login(model);
 }
 }
}
```

```

 }
 return View();
}

```

In this example, if the user adds their email address during registration, an **Email** type claim will be created with the same value as the user's email.

## Creating a Claim Policy

To authorize for claims, you will need to create a simple authorization policy in your application. An authorization policy is a ruleset which is validated whenever the policy is invoked. For example, to require a claim to exist, you will need to add a new policy which checks the user for the presence of the claim.

In order to create a policy, you will need to add a call to the **AddAuthorization(\*authorization options action\*)** method on the **IServiceCollection** parameter in the **Startup** class. The authorization options action accepts a single **AuthorizationOptions** parameter, which allows you to call the **AddPolicy(\*policy name\*, \*policy action\*)** method to add a new policy.

The policy name provided to **AddPolicy** determines how you will refer to it when calling it from the **Authorize** attribute, while the policy action is the code that will be executed whenever the policy is validated. It will receive an **AuthorizationPolicyBuilder** parameter which is used to set up requirements for the policy. You can add claim requirements for a policy by using the **RequireClaim(\*claim name\*, \*valid values\*)** property of the **AuthorizationPolicyBuilder** object. The claim name is the name chosen for the claim itself, as defined by creating a claim, while the values are an optional array of values to require when validating the claim. If it is empty, only the presence of the claim will be checked.

The following code is an example of adding a claim-based policy:

## Adding a Claim-Based Policy

```

public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();

 services.AddDbContext<AuthenticationContext>(options =>
 options.UseSqlite("Data Source=user.db"));

 services.AddIdentity<WebsiteUser, IdentityRole>()
 .AddEntityFrameworkStores<AuthenticationContext>();

 services.AddAuthorization(options =>
 {
 options.AddPolicy("RequireEmail", policy =>
 policy.RequireClaim(ClaimTypes.Email));
 });
}

```

In this example, you can see that an authorization policy named "**RequireEmail**" is added. It will authorize all users who have an "**Email**" claim, while blocking users without an "**Email**" claim. If an array of email names was added to the **RequireClaim** function as a second parameter, only the claim values specified would be authorized.

## Using Claims-Based Authorization

After you set up the authorization logic, you can then use the **Authorize** attribute to enforce your claim requirement. By providing a **Policy** parameter to the attribute, you can supply it with a policy name and all calls to the method or controller which are decorated by the attribute will require the claim to be made.

The following code is an example of authorization based on a claim:

### Claim-Based Authorization

```
[Authorize(Policy = "RequireEmail")]
public IActionResult Index()
{
 return View();
}
```

In this example, you can see that the **RequireEmail** policy is enforced on this action. Only users with an **Email** claim will be able to access this method.

### Custom Policy-Based Authorization

In addition to the previous methods, you can also create your own dynamic policies. These are considerably more complex to create, but in exchange offer you the option to create any policy you require. Custom policies can receive parameters to validate against, but in exchange need to be configured from scratch.

Should you require a more specific validation, you can read how to set it up here:

<https://aka.ms/moc-20486d-m11-pg1>

### Using Multiple Authorization Methods

Finally, you can combine multiple different authorization methods, by adding multiple **Authorize** attributes. Whenever more than a single **Authorize** attribute is present on a controller or action, it will only be accessible if all of the authorization requirements are fulfilled.

The following code is an example of multiple authorization attributes:

### Multiple Authorizations

```
[Authorize(Policy = "RequireEmail")]
[Authorize(Roles = "Administrator")]
public IActionResult Index()
{
 return View();
}
```

In this example, the user will only be able to access this action if it has the "**Administrator**" role, and fulfills the "**RequireEmail**" policy.

 **Best Practice:** Whenever an authenticated user attempts to access a resource that they do not have permissions to access, the ASP.NET Core MVC application will redirect the request to "**Account\AccessDenied**". You should add a controller view and method to handle this path, and update the users that they do not meet the criteria to view the page. If you do not do this, users will encounter a browser error.

## Demonstration: How to Authorize Access to Controller Actions

In this demonstration, you will learn how to set up Authorization in the Startup class and how to use simple authorization in the StudentController class.

## Demonstration Steps

You will find the steps in the section "Demonstration: How to Authorize Access to Controller Actions" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD11\\_DEMO.md#demonstration-how-to-authorize-access-to-controller-actions](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD11_DEMO.md#demonstration-how-to-authorize-access-to-controller-actions).

## Lesson 3

# Defending from Attacks

Web applications are designed from the ground up to be accessible. This is because inherently web applications are intended to allow users to access them from multiple different browsers without being reliant on installations. However, this also means that web applications also have to be particularly wary against malicious attacks from a variety of sources.

In this lesson, you will learn about a variety of different security attacks which frequently target ASP.NET Core MVC applications, as well as ways to utilize the ASP.NET Core environment to defend your website and your users.

There are many types of attacks which can target web applications and different kinds of attacks may target different victims. Some attacks specifically attempt to disrupt the web application server, affecting data integrity or disrupting performance, while others may attempt to instead attack the individual user, steal their data, abuse their permissions, or disrupt their experience.

After finishing this lesson, you will be aware of some of the more common types of attacks directed towards web applications and have some basic knowledge in how you can help to mitigate their effects on your application and its users.

### Lesson Objectives

After completing this lesson, you will be able to:

- Understand cross-site scripting and how to defend against it.
- Understand cross-site request forgery and how to defend against it.
- Understand SQL injection and how to defend against it.
- Understand how to support cross-origin requests in your applications and when to allow them.
- Understand HTTPS and SSL and know how to enforce it in your application.

### Cross-Site Scripting

Cross-site scripting commonly known as XSS, is a security exploit by which a malicious user (commonly referred to as attacker) utilizes the input on a website to insert malicious scripts, such as JavaScript code, with the intention of altering the default behavior for a website. When these scripts are run, they can be used to steal private information such as session tokens or cookies and alter the behavior of pages, potentially redirecting users to malicious websites.

To enact this exploit, the attacker will look for an input with content which is later viewable by other users, for example a chat box in a chat or a username in any application in which other users can be seen. When a script is successfully inserted by the attacker, opening a webpage with the script will cause it to run, potentially causing it to steal information from the user or disrupt their ability to use the website.

- Cross-site scripting involves:
  - Inserting malicious code in the session of a user
  - Posting information to other websites without the knowledge of the concerned users
- You can prevent cross-site scripting by:
  - Using the **JavaScriptEncoder** class and the **Encode** method when encoding inputs into JavaScript
  - Using the **URLEncoder** class and the **Encode** method when encoding inputs into URL

Note that any modifications which occur only within data displayed to the attacker and only affect the specific browser it is made from is not an XSS attack. XSS attacks are only such when it affects a browser other than the browser in which they were performed. It is completely impossible to prevent a user from sabotaging the browser which he is using, as most browsers give knowledgeable users full control over the HTML and JavaScript. If the user can affect a different browsing session, even if it only affects the same credentials, it would be considered an XSS attack.

As a general rule of thumb, you will need to be careful in the following areas whenever handling any data that was input from the user in order to avoid XSS attacks:

- HTML inputs. Never display any untrusted data which may come from another source. This can include data sent from another session, data influenced by headers, or even data loaded from databases.
- HTML elements. Putting untrusted data into an HTML element can cause scripts to be run, resulting in malicious activities. This form is particularly easy to abused, as it's very easy to write a script inside of an element.
- HTML attributes. A clever attacker can utilize a standard attribute structure to end an existing element and add a script or can abuse one of the **on-** events of HTML elements to call a function.
- JavaScript. Untrusted data added or inserted in your JavaScript could result in undesirable function calls or undesired code execution.
- Query string. A further vulnerability point could be the query string. In an unsecure environment, data parsed from inside the query string could result in script executions.

You can avoid these issues by ensuring you use appropriate encoding on untrusted input. Encoding ensures that various symbols commonly used in HTML and JavaScript, such as '<', or '\' are replaced with string keys which are globally recognized by browsers and cannot be used to execute code. For example, in HTML, the '<' character will, for example, be replaced with "**&lt;**" while the '\' character will be replaced with "**&bsol;**". All encoded characters follow a specific format, for example HTML encoding begins with '**'** and ends with '**';**', while JavaScript encoding utilizes a "**\u\*key code\***" format. This ensures that browsers will be able to identify that these elements are encoded and render them appropriately.

Furthermore, HTML itself requires encoded characters to display them. This is due to it being impossible to use several characters due to being used as part of the HTML markup. For example, if you place the text "**&lt;This text appears between lesser than and greater than&gt;**" inside of a view, it will render as "**<This text appears between lesser than and greater than>**". If you try and input the completed sentence directly, it will instead create a nonstandard element "**<this>**" with the remaining words flagged as attributes. The first will be perfectly legible and safe code, while the second could present a security exploit.

You can find a list of various standard character encodings in this link: <https://aka.ms/moc-20486d-m11-pg2>

## HTML Encoding in Razor Pages

By far the simplest case to handle, by default, the @ directive when used in the HTML context of a view, will always encode strings by default. This means that you do not need to worry much about using external inputs within Razor. Therefore, the HTML context is secure by default in regards to XSS. It is however crucial to note that this is specifically for HTML and does not affect JavaScript tags within the HTML.

The following code is an example of Razor HTML encoding:

### Razor HTML encoding

```
@{
 var scriptTag = "<script type='text/javascript'>alert('XSS')</script>";
}

@scriptTag
```

In this example, the view will print out the text: "**<script type='text/javascript'>alert('XSS')</script>**". It will not issue a JavaScript alert and no XSS will occur.

 **Best Practice:** It is possible to allow non-encoded HTML to be used within a Razor page, but you will need to make a dedicated effort by using the **HtmlString** class which is not encoded and as such is prone to XSS attacks. It is advised to not allow non-encoded HTML.

### JavaScript Encoding in Razor pages

Unlike the HTML context of Razor pages, using a directive within JavaScript tags is not automatically encoded. This means that you need to be extra careful when trying to inject unknown data into a JavaScript function.

You can get around this issue by injecting the **JavaScriptEncoder** class. The **JavaScriptEncoder** class is a class designed to safely perform encoding on JavaScript objects, preventing malicious code from being able to execute functions. It is accessible by importing the **System.Text.Encoding.Web** namespace. It provides the **Encode** function which encodes JavaScript, ensuring that the desired text will be displayed safely, while not allowing for undesired functions to be invoked.

The following code is an example of JavaScript based XSS:

### JavaScript XSS

```
@{
 var script = "xss()";
}

<script type="text/javascript">
 function xss() {
 alert("XSS");
 }
 document.write(@script);
</script>
```

In this example, when we attempt to write unknown data, the **xss** function is called instead. Depending on the application many undesirable flows can occur. This is highly problematic since any available function could end up being called leading to a large amount of possible issues, from incorrectly rendering pages, all the way to creating invalid links.

The following code is an example of safe JavaScript encoding:

### Safe JavaScript encoding

```
@using System.Text.Encodings.Web;
@inject JavaScriptEncoder encoder;

{@
 var script = "xss()";
}

<script type="text/javascript">
```

```

function xss() {
 alert("XSS");
}
document.write("@encoder.Encode(@script)");
</script>

```

In this example, the string "`xss()`" will be displayed and no functions will be called. This will allow you to ensure correct script behavior for users and prevent malicious data from affecting others.

## Injecting Encoders

Occasionally, you may wish to work with encoding on the controller. You can do this by injecting **HtmlEncoder**, which is used for encoding for HTML content such as element content or attributes, and **JavaScriptEncoder**, which is used for injecting encoded elements into JavaScript code. This can allow you to pre-encode information from unknown sources and not have to encode it on the Razor page itself.

The following code is an example of injecting encoders into a controller:

### Injecting Encoders:

```

private JavaScriptEncoder _javaScriptEncoder;
private HtmlEncoder _htmlEncoder;

public HomeController(HtmlEncoder htmlEncoder, JavaScriptEncoder javaScriptEncoder)
{
 _htmlEncoder = htmlEncoder;
 _javaScriptEncoder = javaScriptEncoder;
}

public IActionResult Index()
{
 const string XSSScript = "<script>alert('XSS')</script>";
 List<string> encodedScripts = new List<string>();
 encodedScripts.Add(_htmlEncoder.Encode(XSSScript));
 encodedScripts.Add(_javaScriptEncoder.Encode(XSSScript));
 return View("index", encodedScripts);
}

```

In this example, you can see that an unsafe text is being encoded into both HTML encoding and JavaScript encoding.

 **Note:** Razor pages utilize **HtmlEncoder** by default. Therefore, while it will always prevent invalid HTML, it can cause errors or security exploits while using the `@` directive in JavaScript.

## Encoding URL Parameters

Occasionally, you might have a situation where you build a URL while using a query string that includes unknown input. As with all other things, this can potentially have changed URLs or problematic query strings, which can potentially change your URL path or modify values of controls bound to them and create undesirable additional controls that benefit attackers. As such, you should always ensure you encode query strings while creating them in the controller.

The following code is an example of a potential URL XSS attack:

### URL XSS Attack

```

public IActionResult RemoveUser(string userName)
{
 string url = string.Format("~/RemovedUser/{0}", userName);
 return Redirect(url);
}

```

In this example, if the username matches a valid relative path in the application, it can potentially cause serious problems. For example a user name such as "`../../Account/Logout`" would actually redirect the current user to a log out method, requiring them to log back in or even worse, the username could potentially be "`../Admin`" potentially deleting an administrator's user.

The following code is an example of using URL encoding to prevent URL XSS:

### Preventing URL XSS with UrlEncoding

```
private UrlEncoder _urlEncoder;

public HomeController(UrlEncoder urlEncoder)
{
 _urlEncoder = urlEncoder;
}

public IActionResult RemoveUser(string userName)
{
 string url = string.Format("~/RemovedUser/{0}", _urlEncoder.Encode(userName));
 return Redirect(url);
}
```

## Cross-Site Request Forgery

Cross-site request forgery (CSRF or XSRF) is a specific type of attack which utilizes users being logged in on one site by adding calls to APIs on that site from another site. This form of attack can be particularly dangerous, as it can be used to perform operations using the logged in user without requiring any consent from the user.

In order to perform this kind of attack, the malicious website will have a call to an API used by the victim site and will use the user's authentication cookie to perform it. When the malicious site calls the API on the victim site, all cookies for that domain will be sent alongside the request. Thus, potentially allowing any action that can be performed by the user, including deleting or changing data, deleting the user, or any variety of other malicious options dependent on the API.

### Cross-Site Request Forgery

- Exploits existing cookies on a browser to perform actions on victim sites maliciously.

To prevent them in your ASP.NET Core MVC applications you should do the following:

- Utilize **ValidateAntiForgeryToken** or **AutoValidateAntiForgeryToken** to protect actions on your controllers
- Use tag helpers and forms with **PUT**, **POST** or **DELETE** methods in your views to ensure tokens are created correctly



**Note:** An XSRF attack does not even require the user to click any buttons. The malicious website can create a hidden form within the HTML and submit it by using JavaScript, without requiring any input from the user.

An example for a potential XSRF attack:

1. The user logs in to "**www.contoso.com**", receives an authentication cookie, and performs various operations.
2. The user then logs on to "**www.adventure-works.com**", which maliciously added an html form with a **POST** method and the action "**http://www.contoso.com/buyContosoApp**".
3. The form is then submitted through JavaScript. The browser sends the request alongside the authentication cookie since the request is to "**www.contoso.com**" and the user has an active authentication cookie.

4. The request is performed on the "**www.contoso.com**" server without the user being aware of it. Any operation the user can perform can occur in this way.

XSRF attacks are not influenced by utilizing a more secure connection using HTTPS, since the form can just send the request by using HTTPS and it can also be used to target any possible method.



**Best Practice:** If your website API provides any **GET** methods which are capable of changing data, malicious users can target your site on websites which allow user images by providing a vulnerable URL to your website causing problems with any of your users which load any of these image links. It is therefore a best practice for a **GET** method to never change data.

There are many additional ways in which an XSRF attack can be performed, not only requiring cookies, and you must take care to protect your websites and applications.

ASP.NET Core MVC utilizes a form of anti-forgery validation method in which when a form is created by the controller as part of a view, it receives an additional hidden input with the name

**\_RequestVerificationToken**. This name is generated every single time the view is created and thus provides a unique key by which to identify the specific instance which posted the form. When XSRF protection is set up, all XSRF protected actions will validate that a correct token is provided or an error will occur instead. This guarantees that requests arrive only from within the application.

## Protecting Controllers from XSRF Attacks

The first step towards protecting yourself from XSRF attacks is by protecting your controller APIs. This can be done by adding the **ValidateAntiForgeryToken** attribute to a controller or action. By doing this, the controller or action will require that a valid token will be present on the request. This helps to ensure that the action can only be accessed from within a valid ASP.NET Core MVC page.

The following code is an example of an action with the **ValidateAntiForgeryToken** attribute:

### ValidateAntiForgeryToken Attribute

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(string userName)
{
 return View(userName);
}
```

In this example, you can see that **post** calls to the **Index** action will require that the user possess a valid anti-forgery token to call this action. Any attempts to access this index from outside a valid form will result in an error instead.

You may sometimes find yourself in need of marking multiple requests across a controller, while requiring some requests to not require anti-forgery, such as various **GET** based actions. There are several tools you can use to help the process become more convenient and comfortable for the user.

The first is the **AutoValidateAntiforgeryToken** attribute. It behaves similarly to the **ValidateAntiForgeryToken** attribute; however, it will automatically ignore actions which are called with the methods: **GET**, **HEAD**, **OPTIONS**, and **TRACE**, which are designed for data retrieval. This allows you to quickly and easily add anti-forgery to all methods which can change data, without affecting methods for retrieving data.

The following code is an example of the **AutoValidateAntiforgeryToken** attribute:

### AutoValidateAntiforgeryToken

```
[AutoValidateAntiforgeryToken]
public class AntiForgeController : Controller
{
 public IActionResult Index()
 {
 return View();
 }

 [HttpPost]
 public IActionResult Index(string userName)
 {
 return View("Index", userName);
 }

 [HttpDelete]
 public IActionResult RemoveUser(string userName)
 {
 string url = string.Format("~/RemovedUser/{0}", userName);
 return RedirectToAction("Account", "RemoveUser", "User");
 }
}
```

In this example, the normal **Index** action (**GET**) will work regardless of origin, while both the **Index** action with a **POST** method and the **RemoveUser** action which is a **DELETE** method will both require the client to utilize anti-forgery tokens.

Finally, if you wish to specifically make an action accessible in a controller which requires anti-forgery tokens you can use the **IgnoreAntiforgeryToken** attribute to remove it for a specific action. This will let you make certain actions available in scenarios where they are usually blocked.

The following code is an example of the **IgnoreAntiforgeryToken** attribute:

### IgnoreAntiforgeryToken Attribute

```
[AutoValidateAntiforgeryToken]
public class AntiForgeController : Controller
{
 public IActionResult Index()
 {
 return View();
 }

 [HttpPost]
 [IgnoreAntiforgeryToken]
 public IActionResult Index(string userName)
 {
 return View("Index", userName);
 }

 [HttpDelete]
 public IActionResult RemoveUser(string userName)
 {
 string url = string.Format("~/RemovedUser/{0}", userName);
 return RedirectToAction("Account", "RemoveUser", "User");
 }
}
```

In this example, both **Index** actions will always be accessible, but the **RemoveUser** action will require an anti-forgery token.

## Using Anti-Forgery Tokens on Views

To support anti-forgery tokens inside your views, allowing you to interact with actions which require them, you will need to ensure that your forms support anti-forgery tokens. The simplest way to implement this is by ensuring tag helpers are loaded for the view. The forms will also need to possess the method attribute with the values: **post**, **put**, or **delete**. Forms with the **get** method will not receive a token by default.

When tag helpers are loaded into a view, all forms will also generate an **input** element with a type of **hidden** and the name **\_RequestVerificationToken**. This input will also bear a uniquely generated value which will be sent alongside the form and be used to identify that the request came from a known form. This makes it very difficult for another website to replicate and makes the action more secure against XSS attack.

The following code is an example of an anti-forgery token which will be generated on the view:

### Anti-Forgery on a View

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<form asp-action="Index" method="post">
 <input name="userName" type="text" />
 <button type="submit">Done</button>
</form>
```

In this example, you can see that a form is created to perform a **post** method to the **Index** action. Since tag helpers are enabled, it will be generated with an anti-forgery token.



**Best Practice:** It is always possible to specifically disable Form tag helpers to prevent the form from utilizing anti-forgery tokens. However, it is not recommended since it means you choose not to utilize the protection offered by the anti-forgery mechanism.

Should you find yourself needing to use JavaScript to call your methods, you can create a form and either submit it via JavaScript or you can find the hidden element with the name **\_RequestVerificationToken** and use it to create your request.

If you need to add a secure call to a **get** method, such as to prevent hostile websites from getting data from your API or if you wish to not utilize the form tag helper, you can manually add a directive call to **@Html.AntiForgeryToken()** to create the request token. This can allow an alternative option and help you create secure **get** requests.

The following code is an example of the **Html AntiForgeryToken** directive:

### Html AntiForgeryToken Directive

```
<form method="put">
 <input name="userName" type="text" />
 <button type="submit">Done</button>
 @Html.AntiForgeryToken()
</form>
```

In this example, an anti-forgery token is generated inside the form, tag helpers are not required.

## SQL Injection

SQL injection attacks are a common yet extremely dangerous form of malicious attacks which can be used to affect applications. Unlike the majority of attacks introduced in this chapter, SQL injection attacks can also be used to target applications without any web functionality and as such familiarity with how they work is required for any developer.

SQL injection shares a similarity to XSS in that both attacks use parameters to add logic for performing operations. However, whereas XSS targets other clients in their attacks, potentially affecting other users, SQL injection directly targets the database itself. Similar to XSS, SQL injection occurs when a malicious user calls for APIs or changes inputs. But unlike code designed to run scripts, SQL injection instead manipulates input strings and utilizes them to change how SQL queries work. SQL injection could be caused by users providing malicious control input or even directly modifying query strings and server requests on their browsers.

For example, in a basic scenario you might have a controller which receives an ID parameter from a user and utilizes it to get the username for the provided ID.

The following code is an example of a controller action which is vulnerable to SQL injection:

### SQL Injection Susceptible Action

```
public IActionResult GetUser(string id)
{
 using (SqlConnection connection = new SqlConnection(_connectionString))
 {
 connection.Open();

 using (SqlCommand command = new SqlCommand(string.Format("SELECT * FROM Users
WHERE ID ={0}", id), connection))
 {
 using (SqlDataReader reader = command.ExecuteReader())
 {
 string result = "";
 while (reader.Read())
 {
 result = (string)reader["UserName"];
 }
 return View(" GetUser", result);
 }
 }
 }
}
```

#### SQL Injection Attack

To prevent this attack, you can:

- Utilize Entity Framework or other ORMs – These libraries are designed with SQL injection in mind, and make it harder to perform
- Use parameterized queries – These can allow you to create dynamic SQL while preventing external sources from affecting it
- Use stored procedures – Stored procedures use parameters, and do not execute SQL that is not present inside the stored procedure
- Use the lowest required permissions – Restricting permissions granted to your application can help prevent table modifications
- Sanitize parameters – Ensure that any parameter added into SQL cannot be used to run SQL code

In this example, you can see a basic method which receives a parameter with the string type and adds it into the query string before executing it. However, this code is extremely vulnerable to SQL injection as you can add malicious code to drastically alter the behavior of the query for example:

The following code is an example of malicious input that can result in SQL injection:

### Malicious Input for SQL Injection

```
1; DROP TABLE USERS;
```

When this example is provided to the previous example, whether through putting it into the UI input and submitting, through modifying a query string, or through other means, the query once executed will attempt to completely delete the table **USERS** from the database. If this ever affects any relevant database it can instantly cause massive losses, leading to potentially catastrophic results.

### How to Defend Against SQL Injection

Fortunately, there are many ways to defend against possible SQL injection attacks in your ASP.NET Core MVC applications.

#### Utilize Entity Framework

The simplest approach would be to avoid using SQL commands directly in your code. When using Entity Framework and directly working with entity objects, you are not vulnerable to SQL injections. Since Entity Framework does not use an SQL syntax to work with, any attempts to use SQL injection will be stopped by the Entity Framework and at worst will result in the malicious user receiving error messages.

#### Utilize Parameterized Queries

As part of the SQL command, you can choose to not directly add parameters into your query, but instead add named parameters by using the `@*name*` placeholder, where the name is the relevant name for your application. This allows you to add the parameters by calling the `AddWithValue(*name*, *value*)` method on the `SqlCommand` object's `Parameters` property. The `name` property will need to be the same as the placeholder, while the value is the value which you wish to poll the database. This will not result in any SQL being executed, and at worse, the command will fail if the value cannot be applied.

The following code is an example of parameterized queries:

#### Parameterized Queries

```
public IActionResult GetUser(string id)
{
 using (SqlConnection connection = new SqlConnection(_connectionString))
 {
 connection.Open();

 using (SqlCommand command = new SqlCommand(string.Format("SELECT * FROM Users
WHERE ID = @id", id), connection))
 {
 command.Parameters.AddWithValue("id", id);
 using (SqlDataReader reader = command.ExecuteReader())
 {
 string result = "";
 while (reader.Read())
 {
 result = (string)reader["UserName"];
 }
 return View("GetUser", result);
 }
 }
 }
}
```

In this example, you can see that the user ID is added as a named parameter. As a result, even if malicious SQL is inserted, it will be treated as a string instead of part of the query.

## Create Stored Procedures

Another good option would be to create stored procedures to handle various tasks for you. When you call a stored procedure, you are able to call parameters by name. This will prevent unpredictable SQL being run on your server and ensure that only expected code is run. However, creating stored procedures will require additional work to create and maintain separately from your application.

## Use the Lowest Required Database Permissions for the Application

You can always configure the application connection string to use the lowest required access permissions to your database. This is useful when an SQL injection attack does end up occurring, since it can help to limit potential damage to the database by malicious operations. It is a good idea to utilize this among additional means, but should not be exclusively relied on, as it will usually not help protect against attempts to steal data.

## Sanitize Parameters

Finally, as the least recommended option, you can sanitize parameters either by parsing or by using them as part of the routing. If the parameters are not string type you can parse them yourself or you can rely on ASP.NET Core MVC Routing to provide parameters of the correct types. Alternatively, you can use regular expressions and whitelisting to disallow specific strings from appearing.

## Cross-Origin Requests

Modern browsers utilize a special restriction, known as same-origin policy, which is designed to prevent websites of different origins accessing resources on another website. This helps prevent malicious websites from being able to call APIs on other websites and helps protect both end users, as well as the application itself. However, you may occasionally find a need to have different web applications communicate with each other, despite having a different origin. To allow this, you can enable Cross-Origin Resource Sharing (CORS).

### Cross-Origin Resource Sharing (CORS)

- By default your server will not accept any cross-origin requests from external sources.
- You can enable it and utilize CORS policies to create conditions under which your application content will be accessible to external applications.
- Enabling CORS can create a risk within applications and needs to be handled with care. It is important to create policies as specific as required.

CORS allows servers to determine that they allow some or all of these normally blocked access attempts by allowing the developer to set conditions under which the application will accept requests from another origin. The behavior for CORS is a W3 standard, and thus standard across all major browsers and as such does not suffer from compatibility issues. It is important to note however that when utilizing CORS you do not accidentally open yourself up to various attacks.

## Same-Origin

As covered before, CORS needs to be enabled whenever the request origin is changed. The request origin is determined by matching up the following segments of the URL:

- Schema. The protocol declaration at the beginning of the URL, such as **http://** and **https://**. For example, **http://www.contoso.com** and **http://www.adventure-works.com** both have the same schema. Meanwhile **https://www.contoso.com** bears a different schema.
- Host. The host is comprised of both domains and subdomains and encompasses the section between the schema and the port. For example, **https://www.contoso.com** and **http://www.contoso.com:8080** both share the same host. Meanwhile, **https://contoso.com**, **https://www.adventure-works.com** and **https://www.contoso.com** all possess different schemas.

- Port. The port is an optional part of the URL, appearing immediately after the host when it is present. If the port is not specified the default values for the protocol will be used, such as **80** for **http**, and **443** for **https**. When a port is present it will be denoted by **\*:port number\***. For example, **http://www.contoso.com:80** and **http://www.contoso.com** will both have the same port number, **80**. Meanwhile **https://www.contoso.com** and **http://www.contoso.com:442** have different ports at **443** and **442**, as will **http://www.contoso.com** with **80** and **https://www.contoso.com** with **443**, but **http://www.contoso.com** and **https://www.contoso.com:80** will both use port **80**.

Any parts appearing past the schema, host, and port will not affect the origin, but all three must be identical for it to count as being the same origin. If you have two applications which need to communicate on two separate origins, you will need to implement CORS.

## How CORS Works

CORS relies on a specific HTTP Header being utilized as part of the request and the response to confirm to the browser that a request is intended to be CORS. As part of the request, the **Origin** header needs to appear. This header is considered a "forbidden" header and is specifically set by the browser and cannot be changed manually, through JavaScript, or other means. This prevents users from being able to switch the **Origin** values and calling services from unapproved websites.

 **Note:** If the browser supports CORS, such as Microsoft Edge, the **Origin** header will be added by default. CORS cannot be used in browsers which do not add **Origin** to the request.

The following code is an example of a CORS request header:

### CORS Request Header

```
GET http://www.contoso.com/api/getEmployees HTTP/1.1
Referer: https://www.adventure-works.com
Accept: /*
Accept-Language: en-US
Origin: https://www.adventure-works.com
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/17.17134
Host: https://www.adventure-works.com
```

Note in this example that the **Origin** header is set to "**https://www.adventure-works.com**", while the request is made to "**http://www.contoso.com**". This is set by the browser whenever a request to another origin is made.

When the server receives a CORS request, it will validate that the request is from a legitimate source. If the request came from a legitimate source with a valid **Origin** header, the server will add the header **Access-Control-Allow-Origin** with a value equal to the original request **Origin**. The browser will then ensure that there is a match between both values, and if no match is found, the browser will disallow the request, even if the server returns a valid response.

The following code is an example of a CORS response header:

### CORS Response Header

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: https://www.adventure-works.com
Date: Mon, 31 Dec 2018 23:59:59 GMT
Content-Length: 12
```

As you can see, this example returns the header **Access-Control-Allow-Origin**: <https://www.adventure-works.com>. If it matches the origin, the browser will allow the response to be utilized. If there is any inconsistency, the browser will throw an error instead.

## Registering CORS Services

There are two ways in which you can use CORS in your ASP.NET Core MVC application, either through a single middleware or through attributes in ASP.NET Core MVC controllers. Regardless of which option is right for your application, you will need to add a call to the **AddCors()** method on the **IServiceCollection** parameter of the **ConfigureServices** method. This ensures that the appropriate services are loaded for enabling CORS.

The following code is an example for registering CORS services in **ConfigureServices**:

### Register CORS Services

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddCors();
}
```

## Using CORS in Middleware

The first approach to using CORS in your application is by adding a dedicated middleware to handle CORS requests on a global application level. By taking this approach, you will need to customize which requests you allow and which you deny. This is vital, because if you do not limit access to your application from external applications while using CORS, you open your application for misuse from malicious websites.

In order to add CORS as a middleware you will need to add a call to the **UseCors** middleware. The **UseCors** middleware can be called with the **AddCors(\*cors policy builder\*)** to create a policy on which CORS will rely. This policy builder supports various methods with which to customize your policy. One of the most basic yet effective among them is the **WithOrigins(\*origin array\*)** method. It accepts an array of possible origins present as strings as covered earlier. When a request is made from an allowed origin it will be allowed to continue. However, if the origin does not match any of the options, the server will block the request and the browser will display an error.

 **Note:** Note that origins should not end in a trailing '/'. If they do, they will not be parsed correctly.

The following code is an example of the **UseCors** middleware with a policy builder:

### UseCors with Policy Builder

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseCors(builder =>
 {
 builder.WithOrigins("https://www.contoso.com");
 });

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Hello World!");
 });
}
```

In this example, if any cross-origin requests occur from "<https://www.contoso.com>" the server will serve them the same as if they came from within the website. Requests from any other site will not be accepted and result in an error.

An alternative option is also present by instead defining a CORS policy in the **AddCors** method. This can be done by invoking it as **AddCors(\*cors options action\*)**. This method accepts a CorsOption object which can be used to create multiple CORS policies by invoking the **AddPolicy(\*policy name\*, \*cors policy builder\*)** method. This will create a named CORS policy which can later be used in the application, such as within the middleware. To call it from the CORS middleware, use the method call **UseCors(\*policy name\*)** which will receive the policy name declared earlier.

The following code is an example of creating a CORS policy:

### Creating a CORS Policy

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();
 services.AddCors(option =>
 {
 option.AddPolicy("FromContoso", builder =>
 {
 builder.WithOrigins("https://www.contoso.com", "http://www.contoso.com");
 });
 });
}
```

This policy is named "**FromContoso**" and will accept both "<https://www.contoso.com>" and "<http://www.contoso.com>". Note that at this point the policy is created, but will not be applied.

The following code is an example of using CORS policy to set up the CORS middleware:

### Using CORS Policy in Middleware

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseCors("FromContoso");

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Hello World!");
 });
}
```

The "**FromContoso**" policy will be set up as middleware for the application.

 **Note:** It is important that the call to the **UseCors** middleware is made before any resources which may depend on CORS are called. If the call to **UseCors** appears after **UseMvc**, any ASP.NET Core MVC logic will ignore CORS.

### CORS in MVC Controllers and Actions

As an additional option in utilizing CORS is the ability to control CORS on a per action or controller level by utilizing the **EnableCors(\*cors policy\*)** attribute on the action or controller. It utilizes CORS policies which are created as part of the service registration and as such you can have multiple policies in effect for different areas within your application.

The following code is an example of enabling CORS on an action:

### Enabling CORS on an Action

```
[EnableCors("FromContoso")]
public IActionResult Index()
{
 return View("index");
}
```

In this case, the "**FromContoso**" policy will be applied on any requests to the action.

In the event you wish to disallow CORS on a specific action, you can use the **DisableCors** attribute on a specific action inside a controller implementing **EnableCors**. This will prevent CORS from being used on the specific action.

The following code is an example of disabling CORS for an action:

### Disable CORS for an Action

```
[EnableCors("FromContoso")]
public class HomeController : Controller
{
 [DisableCors]
 public IActionResult Index()
 {
 return View("index");
 }
}
```

In this example, any attempts to call the **Index** action from external sources will be blocked.

### Additional CORS Policy Options

As part of **CorsPolicyBuilder** you can use a wide variety of optional methods to configure which servers have permissions to access the application. You can use multiple methods in the same policy to create more specific policies and as a result provide a safer solution.

Several of the methods available to you include:

- **WithOrigins(\*origins\*)**. Expects an array of origins. All origins specified will be able to perform CORS operations.
- **AllowAnyOrigin()**. Allows all origins to access via CORS.
- **AllowAnyMethod()**. Allows using all HTTP methods. Note that without this setting, some types of request methods such as **OPTIONS** may fail.
- **WithHeaders(\*headers\*)**. Expects a list of accepted header values. At least one must be present in the request.

The following code is an example of a custom CORS policy:

### Custom CORS Policy

```
option.AddPolicy("CustomCORS", builder =>
{
 builder.WithOrigins("https://www.contoso.com")
 .AllowAnyMethod()
 .WithHeaders("my-header");
});
```

In this example, the policy will accept CORS requests from "**www.contoso.com**", and from all the HTTP method types. The header "**my-header**" will also have to be present in the request.

## Secure Sockets Layer

HTTP is not a secure protocol. Data transferred through HTTP is not encrypted and can be intercepted and changed by entities residing as part of the pipeline. These entities can include, but are not limited, to ISPs, various temporary internet providers such as hotels, or even malicious attackers throughout the internet.

While utilizing HTTP these entities can change data passing through the sites, whether to inject their own content instead of existing content, purposefully alter or steal data, or even redirect users to web sites completely under their control.

### SSL:

- Encrypts content by using the public key declared by the certificate
- Decrypts content by utilizing the private key only available to the certificate owner
- Ensures that data sent across the web is encrypted, making it much harder to steal or change
- Certificate is determined on the initial request between the client and server

This presents a serious danger to both the user and the application and you can work to help reduce these effects by using the more secure HTTPS protocol. The HTTPS protocol utilizes the Secure Sockets Layer (SSL) protocol on top of an HTTP infrastructure to encrypt data sent between a client and a server. This is done by utilizing approved security certificates to provide hashing, which is then used to encrypt and decrypt the data that is sent.

The security certificates comprise of both a private and public key. When the initial connection is made the client and the server both agree on a security certificate which is approved by a certificate authority (CA) which is trusted by both the client and the server. There are multiple security authorities in charge of issuing certificates and certificates are usually released for individual websites, often with details specifically involving those sites. This is done to prevent communication being open for malicious attacks. The client will be able to access a public key for the site which can be used for performing the encoding on the information, while only the server has access to the private decryption key.

If the client and the server do not have any matching certificates, communication will not occur, as a secure connection cannot be established.

By utilizing SSL, you can help improve application security and reduce the ability of malicious proxies or attackers to intercept your code.



**Note:** It is important to note that regardless of how well you protect your application from running HTTP requests, if the user attempts to access your application by using HTTP, any details sent by the user can be intercepted.

## How to Enforce HTTPS in ASP.NET Core MVC

In ASP.NET Core MVC, you can fairly easily ensure that your users utilize HTTPS while using your application. This can be done by adding the **UseHttpsRedirection()** middleware inside your application. By adding a call to it on the **IApplicationBuilder** object within the **Configure** method, the server will be configured to redirect any requests that reach the server's HTTP port to HTTPS. This ensures that users inside your application are using HTTPS for communication and not HTTP.

It is important to note that the call to the **UseHttpsRedirection** middleware must appear before any middleware which can provide the user with any html files, resources, or actions. This includes, but is not limited to, **UseMvc** and **UseStaticFiles**. If they are called before **UseHttpsRedirection**, these resources will be accessible to normal HTTP traffic.

The following code is an example of enabling HTTPS redirection inside your applications:

### Enabling HTTPS Redirection

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseHttpsRedirection();

 app.UseStaticFiles();

 app.UseMvc();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Hello World!");
 });
}
```

### Setting up the Development Certificate

During development and while testing deployment, you may find that you do not always have an available certificate from a trusted source. Obtaining a certificate is an expensive process requiring you to select a certificate authority and buy a license to use a certificate. This is a required part of a product when it is released, but it is not usually feasible for testing purposes, which may occur years before the application is released to customers.

In order to enable testing your application on deployments, the .NET Core SDK which is used for running ASP.NET Core application adds a special certificate, the "**ASP.NET Core HTTPS development certificate**" on the server on which it is installed. You can then add the certificate to the trusted certificate list for the machine by using the .NET CLI. You will learn more about the deployment process and the .NET CLI in Module 14, "Hosting and Deployment".

The command for adding the development certificate within the .NET CLI is: **dotnet dev-certs https --trust**. When you type it on a server with the Microsoft .NET Core SDK installed, it will be added to the trusted certificates and that machine will be able to access development builds on a browser.

The following code is an example of installing the ASP.NET Core HTTPS development certificate:

### Installing ASP.NET Core HTTPS Development Certificate

```
dotnet dev-certs https --trust
```

# Lab: Managing Security

## Scenario

You have been asked to create a web-based library application for your organization's customers. The application should have a page showing the most recommended books, login and register pages, and the ability to add books to the library for only authorized users. The application should have Identity configuration, a variety of settings for authorization, and a demonstration of a cross-site request forgery attack.

## Objectives

After completing this lab, you will be able to:

- Use Identity.
- Add Authorization.
- Avoid the Cross-Site Request Forgery attack.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD11\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD11_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD11\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD11_LAK.md).

## Exercise 1: Use Identity

### Scenario

In this exercise, you will first add an entity-framework-database context to the **LibraryContext** class. You will then enable using identity in the **Startup** class. After that, you will add sign-in, and register user logic. Finally, you will retrieve data from the identity property in the **LendingBook.cshtml** view.

The main tasks for this exercise are as follows:

1. Add the Entity Framework database context.
2. Enable using Identity.
3. Add sign in logic.
4. Add Register a user logic.
5. Retrieve data from the Identity property.
6. Run the application.

## Exercise 2: Add Authorization

### Scenario

In this exercise, you will first add **AuthorizeAttribute** to the **LibraryController** class. You will then configure role-based and claim-based policy authentication. And, you will add the relevant attribute in the **AccountController** class and in the **LibrarianController** class.

The main tasks for this exercise are as follows:

1. Add the AuthorizeAttribute to an action.
2. Add role-based policy authentication.
3. Add claim-based policy authentication.
4. Run the application.

## Exercise 3: Avoid the Cross-Site Request Forgery Attack

### Scenario

In this exercise, you will first write the Cross-Site Request Forgery attack in a separate project. You will then run the application and see the possible attack. Finally, you will avoid the Cross-Site Request Forgery attack by adding the **ValidateAntiForgeryToken** attribute in the **AccountController** class, run the application, and see that the attack is not possible.

The main tasks for this exercise are as follows:

1. Write the Cross-Site Request Forgery attack.
2. Run the application – Now the attack is possible.
3. Avoid the Cross-Site Request Forgery attack.
4. Run the application – Now the attack is not possible.

**Question:** A member of your team needs to configure the database context to work with identity dbContext, how should he do it?

**Question:** A member of your team changed the **LibrarianController** class, removed the following attribute from the class **[Authorize(Roles = "Administrator")]** what is the impact of his change?

# Module Review and Takeaways

In this module, you learned about how to set up authentication and authorization, and how ASP.NET Core MVC can help you defend against attacks. You can utilize authentication and authorization to help you limit users to parts of your application they have permissions for, while preventing them from accessing sensitive data. You have also learned about many different attacks and security exploits which are frequently used against web applications, and how you can utilize ASP.NET Core MVC applications to better protect both your application and innocent users.

## Review Question

**Question:** What would be the risk if you add a call to an action using HTTP instead of HTTPS?

## Best Practice

Remember that the methods outlined in this module exist to help you protect your application, but will not necessarily stop all malicious attack. You should always be vigilant and work to remove any holes in your application security if they are discovered.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
While running an application with <b>UseHttpsRedirection</b> middleware configured, you manage to navigate to a page using HTTP.	
Users are unable to access pages with a policy requiring administrator permissions, even if they are administrators.	



# Module 12

## Performance and Communication

### Contents:

Module Overview	12-1
Lesson 1: Implementing a Caching Strategy	12-2
Lesson 2: Managing State	12-11
Lesson 3: Two-Way Communication	12-20
Lab: Performance and Communication	12-32
Module Review and Takeaways	12-34

## Module Overview

Modern web applications require complex interactions with users. Users will often request a lot of data in a small time-frame, while also expecting relevant data as soon as it comes online. This can easily cause a significant amount of load on an unprepared server, resulting in unnecessarily complex or repeated operations and a heavy load on your server. Fortunately, there are multiple ways to reduce the load.

Caching allows you to store commonly repeated requests, preventing the need to perform the same logic repeatedly. By using caching, you can reuse data that has already been loaded and present it to the user. This provides the user with a fast response time and reduces system resources used in conducting the logic for the action.

State meanwhile allows achieving a state of consistency between different requests. By utilizing various forms of state management, you can transform the normally stateless web experience into one that is custom tailored to individual clients, with different users enjoying a separate and relevant experience in the same application.

Finally, SignalR is a framework that allows the abstraction of several different communication protocols into an easy to use API, which allows you to easily create a single set of tools on the server and client to facilitate two-way communications. This allows you to focus on the logic you wish to implement while allowing you to not have to cater to specific browsers.

### Objectives

After completing this module, you will be able to:

- Implement caching in a Microsoft ASP.NET Core application.
- Use state management technologies to improve the client experience, by providing a consistent experience for the user.
- Implement two-way communication by using SignalR, allowing the server to notify the client when important events occur.

## Lesson 1

# Implementing a Caching Strategy

Usually, web applications display information on a webpage by retrieving the information from a database. If the information that should be retrieved from the database is large or involves complicated logic, the application might take longer to display the information on a webpage. ASP.NET Core MVC supports some caching techniques to help reduce the time required to process a user request.

Before implementing caching, you should first analyze if caching is relevant to your application because caching is irrelevant to webpages whose content changes frequently. To successfully implement caching in your web application, you need to familiarize yourself with the various types of caches, such as the **cache** tag helper, and data cache.

Additionally, in larger-scale applications where scaling will be required, you will need to consider using a distributed cache, allowing multiple separate servers running your application to use the same cache.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of using caching.
- Describe the **cache** tag helper.
- Describe the in-memory cache.
- Describe distributed caching.
- Configure caching.

### Why Use Caching?

Caching involves storing information in the memory of a web server, this information can include but isn't limited to rendered pages, the results of calculations, and retrieving information from a database. If the content that is rendered does not change much, you can choose to store it in the server memory or on distributed caching servers. When a user performs a request to content that is cached, the server will send the cached content instead of repeating costly operations, allowing the server to reply in a more time-sensitive way.

#### Caching:

- Helps improve the performance of a web application by reducing the time needed to process a webpage
- Helps increase the scalability of a web application by reducing the workload on the server
- Can be customized to retain an appropriate life time and priority for the data being cached

There are many advantages of using caching, which include:

- Reduces the need to repeatedly retrieve the same information from the database.
- Reduces the need to reprocess data, if a user repeats the same request multiple times.
- Helps improve the performance of a web application, by reducing the load on servers.
- Helps increase the number of users who can access the server farm simultaneously.

However, you should consider several important details before implementing caching. If you perform caching for content that frequently changes, you will quickly end up in situations where users are receiving irrelevant content. When performing the caching, the data is stored in the web server and used instead of the most recent data existing. This can be highly useful for things like maintaining a shopping cart for the user or for rendering collections of strings which are calculated the first time but remain consistent. However, data that changes regularly, such as various real-time data such as product inventories and areas where the content changes between visits.

Caching can also be handled on environments running across multiple servers. When working with multiple hosting servers, you will need to either work with a sticky session, which means that the same server handles all requests from the same client, or use a distributed cache, which can handle caching for multiple servers in a separate dedicated server. Otherwise, it will result in multiple servers handling separate requests with each server creating its own cache, which results in a waste of memory.

Servers have a limited amount of free memory they can allocate towards caching at any given moment in time. This means that sometimes the server will need to clear out older cache entries on its own. To assist the server in handling these issues, you can set both the caching priority and the caching lifetime. By setting a specific lifetime, you can ensure that cached data doesn't remain for too long, particularly in systems where the cached data refreshes infrequently. By setting caching priority, you can determine that if there are memory issues, lower priority cached data will be deleted first, whereas higher priority cached data will remain unless there is no lower priority cached data.

Caching lifetime can include customization options such as:

- Setting cache expiry time in absolute units of time.
- Setting a date at which the cache will be cleared.
- Setting a period of time for deletion since the cache is last accessed.
- Setting if the expiration is sliding expiration, and the timer is reset whenever the cached resource is accessed.

Caching priority can be set to **Low**, **Normal**, **High**, or **NeverRemove** to determine priority for removal whenever memory is cleaned up.

## Cache Tag Helper

One the easiest ways to cache in an ASP.NET Core MVC application is by using the **cache** tag helper. The **cache** tag helper is used to envelop the ASP.NET Core MVC code which is expected to yield repeated results. By adding this tag helper, the code inside will be rendered on the first request by the user, followed by persisting the code rendered from the first call on future calls while the rendered data remains in the cache.

By default, the **cache** tag helper persists for 20 minutes before expiry.

- One way to cache in an ASP.NET Core MVC application is by using the **cache** tag helper
- **cache** tag helper attributes:
  - enabled
  - priority
  - Expiration attributes: expires-on, expires-after, expires-sliding
  - vary-by attributes: vary-by-query, vary-by-cookie, vary-by-route, vary-by-user, vary-by-header, vary-by-

The following code is an example of using the **cache** tag helper

### Cache Tag Helper

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<cache>
 @for (int i = 0; i < DateTime.Now.Second; i++)
 {
 <div>Number of seconds</div>
 }
</cache>
```

In this example, a number of divs will be displayed according to the number of seconds in the current time when the request is made. Any future requests, while the caching remains, will yield the same number of divs.

Additionally, the **cache** tag helper supports a variety of additional attributes that can be used to further customize the caching logic to fit with the requirements.

By using the **enabled** attribute, you can enable and disable caching for specific segments of code. An example for this kind of usage could be a website that displays stocks either on a daily or a current trend. While displaying the daily stock values, you can retrieve the stock values once and render the same values throughout the day. On the opposite end of the spectrum, if you are looking at the current value of stocks, you are able to disable caching enabling new values to be retrieved in the same refresh while using one component for stocks in both pages.

The following code is an example of **cache** tag helper **enabled** attribute:

### Using the enabled Attribute

```
<cache enabled="@ViewBag.IsStoreOpen">
 Today's product is:
 @Model.Name
</cache>
```

You can also use a variety of attributes to specify expiration behavior for the cached data, which include:

- **expires-on**: Accepts a **DateTime** object which determines until when the caching will remain.
- **expires-after**: Accepts a **TimeSpan** object which determines the amount of time to keep the cached data since the original request was made. If a new request is made the caching time will not be changed.
- **expires-sliding**: Accepts a **TimeSpan** object which determines the amount of time to keep the cached data since the previous request was made.

The following code is an example of the various **cache** tag helper expiry attributes:

### Using Expiration Attributes

```
<cache expires-on="@new DateTime(2025, 12, 31, 23, 59, 59)">
 <div>
 @DateTime.Now.ToString("dd/MM/yyyy hh:mm")
 </div>
</cache>

<cache expires-after="TimeSpan.FromSeconds(60)">
 <div>
 @DateTime.Now.ToString("dd/MM/yyyy hh:mm")
 </div>
</cache>

<cache expires-sliding="TimeSpan.FromSeconds(5)">
```

```
<div>
 @DateTime.Now.ToString("dd/MM/yyyy hh:mm:ss")
</div>
</cache>
```

The content of the first **cache** tag helper will expire by 2026 and continue showing the same date until that time. The content of second **cache** tag helper will expire after 60 seconds since it was first accessed, while the last one will expire five seconds after the last request. If requests keep coming within five seconds of each other, the last one will not expire.

An additional attribute that can be used to further customize when caching happens is the **vary by** attribute. This allows you to maintain separate caches for different properties and even use **vary by** on its own to set up your own logic. This allows you, for instance, to vary the cache by various model or **ViewBag** properties.

Several common options for **vary-by** are as follows:

- **vary-by-query**: Accepts single or multiple values separated by commas, which are key in the query strings. The caching will be based on the values of the specified query string keys and a separate cache will be used for every different value.
- **vary-by-cookie**: Accepts single or multiple values separated by commas, which are cookie names. The caching will be based on the values of the specified cookies and a separate cache will be used for every different value.
- **vary-by-route**: Accepts single or multiple values separated by commas, which are route data parameter names. The caching will be based on the values of the route data parameter names and a separate cache will be used for every different value.
- **vary-by-user**: Accepts the strings **true** and **false**. The caching will be based the currently logged in user and include non-authenticated users. Logging out or logging in will clear the cache.
- **vary-by-header**: Accepts single or multiple values separated by commas, which are field names in the request header. The caching will be based on the values of the specified request headers and a separate cache will be used for every different value.
- **vary-by**: Allows setting custom string value to be used for caching. Whenever the value changes a separate cache will be used. By concatenating strings, you can create a condition as specific as required.

The following code is an example of **vary-by-query**:

### The **vary-by-query** Attribute

```
<cache vary-by-query="MaxPrice, MinPrice">
 <div>
 @for (int i = 0; i < Model.Count; i++)
 {
 <div>
 @string.Format("{0}: {1}", Model[i].Name, Model[i].Price)
 </div>
 }
 </div>
</cache>
```

In this example, the display for the product list will be cached according to the **MaxPrice** and **MinPrice** keys in the query string. As long as the values for both keys not change, if a cache exists for those values it will be used, otherwise, the code will be recalculated and a new cache entry will be added alongside the previous ones.

The following code is an example of the **vary-by** attribute:

### The vary-by Attribute

```
<cache vary-by="@Model.Id">
 <div>
 @Model.Name
 </div>
 <div>
 @Model.Price
 </div>
</cache>
```

In this example, a new cache will be created whenever a new product is loaded. Whenever a model that is already in the cache is loaded, the cache will be used.

Another crucial attribute for caching is **priority**. By setting this attribute you can determine when the cache should be cleared. The valid values for this attribute are the enum values of the **CacheItemPriority** enum, from the **Microsoft.Extensions.Caching.Memory** namespace, with **Low** being the first to be cleared and **NeverRemove** being the lowest priority to clear due to memory constraints. By default, **Normal** will be used.

 **Note:** Despite being named **NeverRemove**, if the memory issues become severe enough, items tagged as **NeverRemove** can be removed.

The following code is an example of using the **priority** attribute:

### The priority Attribute

```
<cache vary-by="@Model.Id" priority="CacheItemPriority.Low">
 <div>
 @Model.Name
 </div>
 <div>
 @Model.Price
 </div>
</cache>
```

 **Note:** Multiple attributes can be used for the caching logic, enabling powerful and diverse **cache** tag helpers.

## Demonstration: How to Configure Caching

In this demonstration, you will learn how to use a **cache** tag helper and how to create a cache for each loaded product by adding a **vary-by** attribute to the **cache** tag helper.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Configure Caching” on the following page:  
[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD12\\_DEMO.md#demonstration-how-to-configure-caching](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD12_DEMO.md#demonstration-how-to-configure-caching).

## The Data Cache

Caching in ASP.NET Core MVC isn't limited to views. Caching can be of great help inside other components such as controllers. This can be useful to retain frequently repeating data and can be used to greatly improve performance, removing repeated calculations, and attempts to access the database. Unlike the **cache** tag helper, data cache can be useful in multiple pages, leading both to have their own advantages and disadvantages. When creating a web application, it is a good idea to think of the best form of caching for each individual case.

To cache data, you can use the **IMemoryCache** service

- Register the service in the **ConfigureServices** method
- Inject the service throughout the components of your application
- Use the **Set** method to store data in the cache
- Use the **TryGetValue** method to retrieve data from the cache

One of the simplest means of server-side caching is by using the **IMemoryCache** service. This service allows you to create a cache for any type and later retrieve it if it is available.

The first step for configuring the data cache is adding the **IMemoryCache** service inside of the **ConfigureServices** method in the **Startup** class. This can be done by calling the **AddMemoryCache** on the **IServiceCollection** parameter. Once this is done, the **IMemoryCache** can be injected throughout the components of your application.

To store an item in the cache, you can use the **Set** method of the **IMemoryCache** parameter. This method operates on a key and value pair, with the first parameter being a key with the **object** type, and the second parameter being a value of the **object** type. This allows any key or value to be set in the cache.

To retrieve an item from the cache, you can use the **TryGetValue** method of the **IMemoryCache** parameter. It receives a parameter of the **object** type which is the key and a second output parameter of the **object** type, which is the variable that will receive the value from within the cache. This method returns true if the data is cached, or false if it is not.

The following code is an example of using the **IMemoryCache** service:

### Using the **IMemoryCache** Service

```
public class HomeController : Controller
{
 private IProductService _ productService;
 private IMemoryCache _ memoryCache;
 private const string PRODUCT_KEY = "Products";

 public HomeController(IProductService productService, IMemoryCache memoryCache)
 {
 _ productService = productService;
 _ memoryCache = memoryCache;
 }

 public IActionResult Index()
 {
 List<Product> products;
 if (!_ memoryCache.TryGetValue(PRODUCT_KEY, out products))
 {
 products = _ productService.GetProducts();
 _ memoryCache.Set(PRODUCT_KEY, products);
 }
 return View(products);
 }
}
```

Additional methods for using caching include the **GetOrCreate** and **GetOrCreateAsync** methods. These methods will check the key for you and if the cache item with the matching key does not exist, perform the logic for retrieving the data. This can help further simplify the code. **GetOrCreateAsync** should be used if the caching logic is reliant on tasks.

The following code is an example of using the **GetOrCreate** method:

### Using the GetOrCreate Method

```
public IActionResult Index()
{
 List<Product> products = _memoryCache.GetOrCreate(PRODUCT_KEY, entry =>
 {
 return _productService.GetProducts();
 });

 return View(products);
}
```

Sometimes the default caching settings isn't useful to you and you want to customize it further. To do this, you can provide the **Set** method an additional third parameter of the type

**MemoryCacheEntryOptions**. The **MemoryCacheEntryOptions** allows you to set useful parameters such as:

- **AbsoluteExpiration**. The absolute date in which the cache entry expires.
- **PostEvictionCallbacks**. A callback function that will be called when the cache entry is removed.
- **Priority**. The priority of the cache entry.
- **SlidingExpiration**. An offset after which the cache is cleared. This is reset every time the cache is accessed.

The following code is an example of using the **MemoryCacheEntryOptions** parameter:

### Using the MemoryCacheEntryOptions Parameter

```
public IActionResult Index()
{
 List<Product> products;

 if (!_memoryCache.TryGetValue(PRODUCT_KEY, out products))
 {
 products = _productService.GetProducts();
 MemoryCacheEntryOptions cacheOptions = new MemoryCacheEntryOptions();
 cacheOptions.SetPriority(CacheItemPriority.Low);
 cacheOptions.SetSlidingExpiration(new TimeSpan(6000));
 _memoryCache.Set(PRODUCT_KEY, products, cacheOptions);
 }

 return View(products);
}
```

## Distributed Cache

A major issue with regular caching is that it is reliant on a specific web server. In a sticky server scenario, this means that you will need to keep working with the same server, even if the server becomes busy and non-responsive. In a non-sticky server scenario, any attempts to use caching are ineffective, possibly resulting in the same data getting cached across multiple servers.

A distributed cache is instead held in a centralized area and is not affected by individual web servers. Therefore, the cache will persist even when servers are taken down. Additionally, data stored on a distributed cache is used for handling requests from multiple users with the same cache. This can greatly reduce the amount of cached data, leading to less storage needed and offering more comprehensive solutions.

It is important to note that unlike other caching options, distributed caching saves and reads the data as a **byte[]**. Therefore, you will often need to convert data types to use distributed caching correctly.

### Distributed cache:

- Stores shared cache data across multiple users and servers
- Can be configured to work with both SQL and Redis
- Is managed by using the **IDistributedCache** interface to cache information in components such as controllers
- Is managed by using a **distributed-cache** tag helper alongside the **name** attribute in views

### Configuring a Distributed Cache

A distributed cache is used through the **IDistributedCache** interface. However, **IDistributedCache** can be implemented by multiple different providers. As a general rule of thumb, you will provide the actual cache implementation itself in the **ConfigureServices** method by calling for the appropriate method.

A distributed cache can be applied in many different ways, such as:

- SQL Server cache, added by calling **AddDistributedSqlServerCache**. This will connect to a cache on a specially configured SQL Server. It requires a connection string, schema name, and table name in its setup.
- Redis distributed cache, added by calling **AddDistributedRedisCache**. This will connect to a Redis data store, Redis being an open source in-memory data store. It requires configuration and Instance name in its setup.

 **Note:** Redis will not be covered in this course. If you want to find out more details about Redis you can read about it here: <https://aka.ms/moc-20486d-m12-pg1>

### Working with **IDistributedCache**

Once the distributed cache has been configured, you are able to inject the **IDistributedCache** throughout your application **components**. The **IDistributedCache** interface exposes several useful methods for interacting with the cache. Most commonly, the **Set** and **Get** method will be used. There is also an option to use asynchronous variants by calling **SetAsync** and **GetAsync**.

The **Get** method receives a key of type **string** as a parameter and returns a value of type **byte[]** if the key is found. If it is not found, null will be returned instead. It is important to remember you will often need to convert the data returned from **byte[]** to another data type used in the application.

The **Set** method receives a key of type **string**, a value to be stored of type **byte[]** and an optional object of type **DistributedCacheEntryOptions**, which allows setting expiration time for the cache based on sliding expiration, absolute expiration and absolute expiration relative to the current time.

The following code is an example of working with the **IDistributedCache** interface:

### **IDistributedCache Interface Example**

```
public IActionResult Index()
{
 string currentTime;
 byte[] value = _distributedCache.Get(CURRENT_DATE);
 if (value != null)
 {
 currentTime = Encoding.UTF8.GetString(value);
 }
 else
 {
 currentTime = DateTime.Now.ToString("dd/MM/yyyy hh:mm");
 DistributedCacheEntryOptions cacheEntryOptions = new DistributedCacheEntryOptions
 {
 SlidingExpiration = TimeSpan.FromSeconds(60)
 };
 _distributedCache.Set(CURRENT_DATE, Encoding.UTF8.GetBytes(currentTime),
cacheEntryOptions);
 }
 return Content(currentTime);
}
```

### **Working with Distributed Cache Tag Helper**

An additional way to utilize distributed cache is by using the **distributed-cache** tag helper. This tag helper is nearly identical to the normal **cache** tag helper, with a few exceptions.

The main change is that it requires a **name** attribute. The **name** attribute is shared between all instances of the same **name** throughout the application and even across multiple separate clients. This means that if two separate users access the same cache on separate browsers, they will receive the same result. This can help boost performance by a large margin for pages which change on a predictable schedule.

If the **distributed-cache** tag helper is used without setting up a distributed cache, the system will use a fallback, and it will behave like a normal **cache** tag helper.

The following code is an example of a **distributed-cache** tag helper:

### **Distributed Cache Tag Helper**

```
<distributed-cache name="defaultCache">
 @for (int i = 0; i < DateTime.Now.Second; i++)
 {
 <div>Number of seconds</div>
 }
</distributed-cache>

<distributed-cache expires-on="@new DateTime(2025, 12, 31, 23, 59, 59)"
name="expiresOnCache">
 <div>
 @DateTime.Now.ToString("dd/MM/yyyy hh:mm")
 </div>
</distributed-cache>

<distributed-cache expires-sliding="TimeSpan.FromSeconds(5)"
name="slidingExpirationCache">
 <div>
 @DateTime.Now.ToString("dd/MM/yyyy hh:mm:ss")
 </div>
</distributed-cache>
```

## Lesson 2

# Managing State

While developing applications, you may want to create functions that require information to be retained across requests. For example, consider an application in which you need to first select a customer and then work on the order relevant to the customer. HTTP is a stateless protocol. Therefore, ASP.NET Core includes different state management techniques to help store information for multiple HTTP requests. You need to know how to configure and scale state storage mechanisms to support web server farms.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of storing state information.
- List the options for storing state information.
- Configure and use Session state in ASP.NET Core MVC applications.
- Store a client-side information with the HTML5 Web Storage API.

### Why Store State Information?

HTTP is a stateless protocol. This means that every single request is independent of each other. However, sometimes you will want to handle several requests in order and to keep them connected to some degree. For example, when shopping online you will want to be able to find a product, add it to a cart, which may already contain additional products, log in, and add personal details. Every single one of these is handled by a separate request and in a truly stateless environment, it will not be possible.

#### Using states:

- Creates a continuity between multiple different requests
- Allows identifying specific users and using user specific logic
- Is required for handling authentication
- Allows developers to overcome weaknesses of using a stateless protocol

Fortunately, while HTTP itself is stateless you can use various methods to create a state on the server. There are many different methods you can use, each of which works differently and allows you to maintain persistent information across multiple connections. By utilizing these methods, you can create a persistent environment for the user, with actions leading to further changes as they use your application.

Most applications will end up supporting state in some form or another, but it can take many different forms. Common technologies used for state management from the client side include cookies, query strings and data from various HTML controls, such as hidden fields. Meanwhile, in the server,

**HttpContext**, alongside the **TempData** property of the controllers can be used to synchronize different requests from the same client to result in a coherent and seamless experience for the user, with individual technologies used for implementing different requirements as needed.

In addition, the client itself can also maintain a degree of session management by using the HTML5 web storage API. This is a form of session management that is entirely in JavaScript on the client side.

 **Note:** It is important to note that authentication cannot be implemented in ASP.NET Core MVC without utilizing some form of state management. Without any state management, authentication would be required in every single request.

## State Storage Options

There are many different technologies that can be used to maintain the application state.

Throughout your application, you will often need to decide which ones are relevant for you and each has its pros and cons towards storing state.

### Client-Side Options

#### Query Strings

One of the simplest ways to obtain persistence throughout the application is by using query strings in requests. By doing this, you can pass a limited amount of extra information on every

request. While a relatively simple way to preserve session state, it can quickly end up with sending very large amounts of information from multiple pages, as well as manually needing to be added to individual requests. Using query strings works best when you need to add very small amounts of additional state related information. To do this, you just need to append the data from previous requests to your current request.



**Best Practice:** Query strings are not secure, and you should never use them to send secure data such as usernames and passwords. They should only be used for adding minimal amounts of data, as it is easy to use them maliciously.

#### Hidden Fields

Another possibility for retaining data is by using input elements with the type hidden. When included as part of a form, these will be sent whenever requests are made from a form and are frequently useful when trying to create multi-page forms, since a hidden field with an identifying value can be used. A major weakness of hidden fields is that they can be relatively easily maliciously changed by the user, requiring validation on the server side.

#### Cookies

Cookies are a form of client-side data retention that allows the storage of key-value pairs. Unlike the previous methods, cookies for a site are always sent on every request, which has both advantages and disadvantages. On the one hand, as developers, you do not need to explicitly add cookies at any point, but on the other hand, if cookies contain too much data, it can slow down requests to the server.

Cookies are usually limited in size and quantity on browsers, and therefore it is a good idea to use a relatively small amount of focused data. A good example of the kind of data frequently stored in cookies is authentication tokens which are returned to the client after a successful login. The client will then store that token on every request, which the server can use to validate the session for the sender.

It is important to note that cookies can easily be manipulated by knowledgeable users, and thus the server should always validate that the cookies are returning expected results.

Also note that cookies can be set up to remain after the browser is closed and can be customized to persist for different amounts of time, allowing behavior similar to caching.

#### State Storage:

- Allows websites to maintain a more coherent continuous experience
- Involves client-side session management techniques such as:
  - Hidden fields
  - Cookies
  - Query strings
- Involves server-side session management techniques such as:
  - TempData
  - HttpContext.Items
  - Cache
  - Dependency Injection
  - Session state

## Server-Side Options

### **TempData**

One of the first methods to handle session state on the server side is by using **TempData**. This is a special data store which stores items until they are read, at which point they are deleted. The **TempData** is shared across all controllers of the application, allowing you to interact with it between completely unrelated requests.

**TempData** can have items added to it by setting a key, and the items are removed when getting a value for a key. Additionally, you are also able to use the **Keep** method, which will mark a key to not be deleted when getting from it, or the **Peek** method, which allows getting the value of a key without deleting it.

The following code is an example of using **TempData**:

#### **TempData Example**

```
public IActionResult Index()
{
 object tempDataValue = TempData["myKey"];

 if (tempDataValue != null)
 {
 return Content("TempData exists!" + tempDataValue);
 }

 TempData["myKey"] = "Temporary Value";
 return Content("TempData does not exist!");
}
```

In this example, you can see that when the **TempData** does not contain the **myKey** key, it will set it in the **TempData** with the value of the string, **Temporary Value**, and return the string content **TempData does not exist!**. If the key does exist however, it will instead be read and deleted from the **TempData** and instead the string content **TempData exists! Temporary Value** will be returned.

By default, **TempData** utilizes browser cookies as the storage mechanism. It is however also possible to use session-based storage by appending a call to **AddSessionStateTempDataProvider** method to the **AddMvc** method in the **ConfigureServices** method. By doing this, the storage mechanism used for sessions will be used instead of cookies.

The following code is an example of configuring **TempData** to use session storage:

#### **TempData Session Storage**

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc().AddSessionStateTempDataProvider();
 services.AddSession();
}
```

 **Note:** **AddSession** and Session Storage will be covered in more details in topic three, "Configuring Session State"

### **HttpContext.Items**

One way of managing state on the server side is by using the **Items** property of the **HttpContext** object that is used by middleware in your application. The **Items** property is a **Dictionary of object** type keys and **object** type values, allowing you to store and access any property you need to be later used in other middleware.

This method of state management is relevant on a request-specific basis, since this property is part of the request itself. However, it allows you to communicate between different middleware and share important information without having to conduct complex methods repeatedly.



**Best Practice:** When creating middleware intended for specific applications, the use of strings as keys for the **HttpContext.Items** dictionary is very useful and will frequently be a good choice. However, when creating middleware that is intended to be used in more than one application, you should use specific objects as keys. This will ensure that your **HttpContext.Items** do not accidentally clash with those of another middleware.

The following code is an example of using the **HttpContext.Items** collection:

### HttpContext.Items Example

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.Use(async (context, next) =>
 {
 context.Items["Query"] = VerifyQueryString(context.Request);
 await next.Invoke();
 });

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync(context.Items["Query"].ToString());
 });
}

private string VerifyQueryString(HttpRequest request)
{
 if (request.QueryString.HasValue)
 {
 return string.Format("The Query String is: {0}", request.QueryString.Value);
 }
 else
 {
 return "No Query String!";
 }
}
```

In the preceding example, you can see an item on the **HttpContext** parameter being set with the key **Query** in the first middleware, and the resulting value being displayed from a different middleware. If no query string is present, the string **No Query String!** will be displayed in the browser. If a query string is present, the string **The Query String is:** followed by the content of the query string will be displayed.

### Cache

An additional means of state management you can use is caching. Caching can be an efficient means of handling repeated requests and handling reoccurring information. Caching is useful for handling the state for repeating requests, ensuring data consistency.



**Best Practice:** When using distributed caching, you should not store any user-specific information in the cache. This could result in at best invalid information being displayed, and at worst present a major security breach.

## Dependency Injection

Since services can be injected into any component throughout your application, they can also be used for state management. There is no end to possible usages for services in state management, and by using them any desired behavior can be achieved.

## Session State

Another method for managing state is the Session State. This is a built-in mechanism designed to specifically handle sessions between the server and client. By utilizing specific cookies from the client, the server is able to identify the specific client and apply user-specific logic throughout the request. The same session can persist for lengthy periods of time, allowing users to keep using the system without requiring constant authentication.

A session will only be created when at least one value is retained as part of the sessions. Sessions will be specific to individual cookies, which are set to deletion when the browser is closed. This can allow a user to continue interacting with the application in a seamless manner. In the event an expired cookie is received, a new session will be opened for handling it. By default, sessions will persist for 20 minutes, but you can configure it as needed. You can also expire the session early by calling the **Clear** method of the **ISession** interface.

 **Note:** There is no default implementation for clearing sessions when the cookie expires, or when the client closes the browser. If you require this behavior, it will need to be created manually.

In single server scenarios, it is possible to store the session in a memory cache. In multiple servers' scenarios, sessions will be stored by using a distributed cache. This allows your session to work across multiple servers, with the client being agnostic to which server handled the last request, allowing for a convenient user experience.

## Configuring Session State

The session state is a built-in mechanism allowing you to easily access a shared data storage that works on a per session basis. This allows you to couple between a specific client, and data relevant to it, without consistently managing identity on every single request. The built-in mechanism uses the distributed cache on the server, and cookies of the client to manage sessions, without requiring any additional work on your end.

To set up the session state, you will need to call the **AddSession** method of the

**IServiceCollection** parameter in the **ConfigureServices** method. You can optionally supply an **Action** delegate as a parameter, which will be used to set up the **SessionOptions**. The **SessionOptions** includes important properties that allow you to configure session behavior, such as cookie behavior via the **Cookie** property, and how long sessions will be stored on the server via the **IdleTimeout** property.

```
public IActionResult Index()
{
 int? visitorCount = HttpContext.Session.GetInt32(VISIT_COUNT_KEY);
 if (visitorCount.HasValue)
 {
 visitorCount++;
 }
 else
 {
 visitorCount = 1;
 }
 HttpContext.Session.SetInt32(VISIT_COUNT_KEY, visitorCount.Value);
 return Content(string.Format("Number of visits:{0}", visitorCount));
}
```

You also need to call the **UseSession** method on the **IApplicationBuilder** parameter in the **Configure** method. This will set up the **Session** property of the **HttpContext**, allowing it to be used in other middleware or in various components throughout the application. It is important to note that **UseSession** should be called before any middleware that will use the session or you will encounter an **InvalidOperationException** when trying to use the session.

The following code is an example of configuring session state:

### Configuring Session State

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddSession(options =>
 {
 options.IdleTimeout = TimeSpan.FromSeconds(20);
 });

 services.AddMvc();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseSession();

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Not Found");
 });
}
```

### Configuring Session State in Distributed Environments

In order to set up a session state in a distributed environment, you will need to set up a distributed memory cache, in the **ConfigureServices** method. Setting up a distributed memory cache was covered in the first lesson, "Implementing a Caching Strategy".

The following code is an example of configuring session state in a distributed environment:

### Configuring Session State in a Distributed Environment

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddDistributedMemoryCache();

 services.AddSession(options =>
 {
 options.IdleTimeout = TimeSpan.FromSeconds(20);
 });

 services.AddMvc();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseSession();

 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Not Found");
 });
}
```

 **Note:** In the example above, you can see that **AddDistributedMemoryCache** is used. This method technically implements the **IDistributedCache**, but it behaves similarly to **IMemoryCache** and is not distributed. It is useful for creating caching with the intention of adding multiple servers in the future and as a method of quickly testing distributed cache methods on development environments. It does not actually handle distributed caching logic.

## Get and Set Session Values

To use the session in your application, you will use the **HttpContext.Session** static property. This property exposes the session API and allows getting and setting values through a variety of useful methods.

You can use the **Set(\*key\*, \*value\*)** method which receives a key of type **string** and a value of type **byte[]**. It will then store the value on the session until it is overwritten by a future call to **Set** or if the session expires.

You can use the **Get(\*key\*)** extension method which receives a key of type **string** to retrieve a value of type **byte[]** which has previously been set by using the **Set** method. If the value is not present in the session it will return null instead.

An alternative option to **Get** is **TryGetValue(\*key\*, out \*value\*)**, which receives a key of type **string**, and an output value parameter which is of type **byte[]** and returns **true** if the value was found or **false** if it was not found.

Another useful method is **Remove(\*key\*)**, which receives a key of type **string**, and removes the entry from the session if it exists.

Additionally, there are also type-specific variants for **Get** and **Set**:

- **SetString(\*key\*, \*value\*)**. Same as **Set**, but value is **string**, rather than **byte[]**.
- **GetString(\*key\*)**. Same as **Get**, but returns **string**, rather than **byte[]**.
- **SetInt32(\*key\*)**. Same as **Set**, but value is **int**, rather than **byte[]**.
- **GetInt32(\*key\*)**. Same as **Get**, but returns **int**, rather than **byte[]**.

The following code is an example of tracking the number of times a user visits a page in the same session:

### Using the **HttpContext.Session** Property

```
public class VisitCounterController : Controller
{
 private const string VISIT_COUNT_KEY = "Visit_Count";

 public IActionResult Index()
 {
 int? visitorCount = HttpContext.Session.GetInt32(VISIT_COUNT_KEY);
 if (visitorCount.HasValue)
 {
 visitorCount++;
 }
 else
 {
 visitorCount = 1;
 }
 HttpContext.Session.SetInt32(VISIT_COUNT_KEY, visitorCount.Value);
 return Content(string.Format("Number of visits:{0}", visitorCount));
 }
}
```

## Demonstration: How to Store and Retrieve State Information

In this demonstration, you will first see how to configure an ASP.NET Core application to use session state. You will then see how to access the session state to store and retrieve information across multiple controllers. After that, you will see how to navigate between the controllers while inspecting the information stored on the session state.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Store and Retrieve State Information" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD12\\_DEMO.md#demonstration-how-to-store-and-retrieve-state-information](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD12_DEMO.md#demonstration-how-to-store-and-retrieve-state-information).

## Using the HTML5 Web Storage API

One additional option for state management that is entirely client-side based is HTML5 Web Storage API. By using a simple API, you are able to store data on the client side, which can then be used later by your application. These APIs are both specific to JavaScript and cannot be directly interacted with from the server. These forms of storage are specific to individual sites. Attempting to access local storage for a different application will not return any data.

There are two types of storage exposed by the HTML5 Web Storage API, differing by the way in which the data is stored. Each means of storage keeps its own separate data and they will not be shared. The same key strings can be safely used for both.

- Types of storage:
  - Local storage – Persists until removed and is shared between tabs
  - Session storage – Exists for a single tab and removed when it is closed
- Functions which exist in both local storage and session storage:
  - Get – Retrieves a stored value for a key
  - Set – Stores a chosen value for a key
  - Remove – Removes a saved value for a key

- **localStorage**. Items stored with **localStorage** will persist until deleted. They will be relevant between different browser windows and tabs and will still exist when the browser is closed. This is useful with information that does not need to change often and data that needs to last for a long time.
- **sessionStorage**. Items stored with **sessionStorage** are specific to the individual browser tab and will not be accessible from other windows or tabs. It will always be cleared once the tab is closed. This is useful for information relating to the current browsing session and for sensitive information.



**Note:** It is important to note that these APIs store data in browser specific storage. Data stored in one browser will never be accessible to another browser.

These APIs hold a dictionary of key-value pairs. Both the key and value are strings and trying to store any other type will store the **toString** value of that variable.

To use the API itself, you can use the following functions, which exist on both **localStorage** and **sessionStorage**:

- **Set(\*key\*, \*value\*)**. Receives a string key parameter and a string value parameter. The value will be stored under the key in the storage.
- **Get(\*key\*)**. Receives a string key parameter and returns the currently stored value for that key. If there is no value for the key, null will be returned instead.

- **Remove(\*key\*).** Receives a string key parameter and removes that entry from the storage if it exists.

The following code is an example of using html5 **localStorage**:

### HTML5 localStorage Example

```
var storage_key = "num_of_visits";
var numberOfVisitsString = localStorage.getItem(storage_key);
var numberOfVisits = 1;
if (numberOfVisitsString) {
 numberOfVisits = parseInt(numberOfVisitsString) + 1;
}
alert("This page has been visited " + numberOfVisits + " times");
if (numberOfVisits >= 5) {
 localStorage.removeItem(storage_key);
} else {
 localStorage.setItem(storage_key, numberOfVisits);
}
```

In this example, the number of visits is tracked by the **localStorage**. Whenever another visit to the page occurs the number will increase and be stored in the **localStorage**. If the user visits the page a fifth time, the entry will be removed and the cycle will restart. This will persist across browser resets and between tabs.

The following code is an example of using HTML5 **sessionStorage**:

### HTML5 sessionStorage Example

```
var storage_key = "num_of_visits";
var numberOfVisitsString = sessionStorage.getItem(storage_key);
var numberOfVisits = 1;
if (numberOfVisitsString) {
 numberOfVisits = parseInt(numberOfVisitsString) + 1;
}
alert("This page has been visited " + numberOfVisits + " times");
sessionStorage.setItem(storage_key, numberOfVisits);
```

In this example, the number of visits is tracked by the **sessionStorage**. Whenever another visit to the page occurs the number will increase and be stored in the **sessionStorage**. If the user attempts to open the tab in a different tab or window, the values will be separate from the original tab. Upon closing a tab, the data is reset.

 **Note:** If you wish to store an object or array rather than a string, you can parse JavaScript objects into JSON format by using the **JSON.stringify()** function and convert JSON data back into JavaScript data by using the **JSON.parse** function.

## Lesson 3

# Two-Way Communication

In the modern era, it becomes increasingly important to receive immediate updates. These days many websites require frequent interaction between multiple users and frequently changing data. Websites facilitating online games and chatting services, as well as providing real-time data are becoming more and more popular. They require the creation of bi-directional websites, where the server updates the client in addition to the client sending messages to the server. In order to facilitate this, you will need to use methods of achieving two-way communication. First, you will be introduced to the WebSocket protocol, which allows websites to handle two-way communication. Then you will be introduced to the Microsoft SignalR library, which allows you to reduce a lot of the setup process required to set up such environments while offering backwards compatibility with older browsers.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the WebSocket protocol.
- Use SignalR in an ASP.NET Core MVC application.
- Add additional configurations to alter the basic behavior of SignalR to conform to additional requirements.

### The Web Sockets Protocol

A major issue with the HTTP protocol is that it is a one-way protocol. The client makes a request to the server, and in return, the server returns a response. The server has no way of initiating a connection with the client. This meant that in order to manage a website which is updated regularly while the user is on it, such as a chat room, various workarounds were implemented traditionally.

Common resolutions involved sending requests to the server which were kept on the server until the event occurred on the server and it finally sent a response (this is called long polling) or creating an endless loop sending requests at set intervals to get updates.

Both of these methods have large flaws, such as spending resources on keeping multiple requests running, sending and receiving a large number of requests increasing the bandwidth used. These can quickly add up and reduce the server's ability to serve additional users.

A solution to these issues comes in the form of the WebSocket protocol. WebSocket is a method of achieving two-way communications between a server and a client. The client will send connection details to the server, which will then be able to use the existing details to make callbacks in the future. This allows the creation of a proper method of two-way communication.

#### Characteristics of web sockets:

- W3C provides the WebSocket protocol to ensure that browsers support WebSockets as part of the HTML5 implementation
- WebSockets facilitate two-way communication between client and server systems
- WebSockets eliminate the need to re-create requests multiple times
- WebSockets function in a similar manner as traditional network sockets

The WebSocket utilizes the existing HTTP framework to create this binding. When the initial connection for WebSockets is made, the **HTTP Upgrade** header is used. This lets the server know that the WebSocket protocol needs to be used, and as long as the server supports it, the status code 101, which represents switching protocol, will be used. At this point, the client will send its payload and the server will store the client details for future requests.

Once the connection has been made both the client and server can use the existing connection to send messages to each other. This facilitates two-way communication and allows proper communication between both server and client.

The client can also choose to close the connection at specific points, allowing the server to clear up any related resources, and further improve our ability to handle requests. The server can also choose to close connections, such as with a timeout, allowing the server to not retain irrelevant data for long.

The WebSocket protocol is frequently used for real-time websites such as stock sites, gaming sites, and chatting sites, as all of those require frequent updates, often user reliant. For example, a multiplayer game requires tracking the movement of all games, while a chat will often alert users of other users writing messages or joining and leaving channels. Note that for websites with infrequent updates, it may be better to perform a request every so often and the WebSocket protocol is more useful for real-time uses.

 **Note:** While WebSockets is a standard protocol that is defined and required by the W3C (World Wide Web Consortium) as part of a modern browser, older browsers may not support WebSockets and other options will be required. As with many other technologies the use of WebSockets should be considered based on the target audience for your browsers.

## Using SignalR

ASP.NET Core SignalR is a library which helps to create applications that utilize two-way communications throughout your application. SignalR will automatically determine and use optimal techniques for achieving two-way communication, preferring WebSockets when available. However, if the browser does not support WebSockets, other options such as long polling will be used instead. This allows you to create applications without having to worry about the client browser, while still offering a two-way communication experience.

- SignalR enables two-way communications between client and server
- Server Side:
  - Configure SignalR in Startup class
  - Define hubs
- Client Side:
  - Use SignalR Client Library
  - Connect to hubs by using JavaScript

SignalR itself is mainly handled in two sections. The first section, called the hub, is server based. The hub declares various methods to be called by the client, with every method being able in turn to reply to one or more clients. As an example, in a chat application, you will often want one method for chatting with all users of the room and an additional method for chatting with a specific user.

The other section is in the client. It is responsible for connection to a specific hub, invoking methods on the hub, and receiving messages from the hub. This can be implemented in various different ways, but in ASP.NET Core MVC applications it is handled in JavaScript.

 **Note:** By default, SignalR uses JSON format to transfer data, although it also supports the MessagePack binary protocol, which can be enabled and used to transfer data even faster. Take note that enabling it will require additional steps to parsing it on the client.

## SignalR in the Server

### Configure SignalR

The first step in adding SignalR to an application is to add the necessary configuration in the **Startup** class. First, in the **ConfigureServices** method, on the **IServiceCollection** parameter, add a call to the **AddSignalR** method.

Second, in the **Configure** method, on the **IApplicationBuilder** parameter, add a call to the **UseSignalR** method. This method expects an **Action** that has an **HubRouteBuilder** generic parameter. This parameter allows you to create routes for the various hubs in use throughout your application.

The following code is an example of configuring SignalR on the server:

### Configuring SignalR on the Server

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddSignalR();
 services.AddMvc();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
 app.UseStaticFiles();

 app.UseSignalR(routes =>
 {
 });
 app.UseMvcWithDefaultRoute();

 app.Run(async (context) =>
 {
 await context.Response.WriteAsync("Hello World!");
 });
}
```

In this example, you can see the basic method calls required to set up SignalR. Currently, no routes are defined since no hub has been created yet.

 **Best Practice:** It is a good idea to set up SignalR middleware before setting up ASP.NET Core MVC middleware in the pipeline. This ensures SignalR related routes will not accidentally be sent to the ASP.NET Core MVC framework.

### SignalR Hubs

The next step is to set up a server hub. This can be done by adding a new class which inherits from the **Microsoft.AspNetCore.SignalR.Hub** class. This class will implement the various hub methods. Generally, hub methods will be **async** methods that return a **Task**. The only restriction on parameters is that they have to be serializable, but any number of parameters can be used.

Inside the hub methods, you will want to also send messages to the various clients. This can be done by using the **Clients** property. This property exposes a variety of possible client options you can send to, all of which implement the **IClientProxy** interface. This interface exposes the **SendAsync** method, which allows various overloads of sending parameters. The most common utilization for it is **SendAsync(\*method\*, \*arg1\*, \*arg2\*.... \*argN\*)**. All parameters are objects which are serialized into JSON or MessagePack and then sent to the various clients. Not every method in the hub will necessarily call the **SendAsync** method, but most will.

The various client properties and methods implementing the **IClientProxy** include:

- **All**. Property which returns all clients connected to the hub.
- **Caller**. Property which returns the client which invoked the hub method.
- **Others**. Property which returns all clients other than the one which invoked the hub method.
- **AllExcept**. Method which receives a list of string connection IDs of clients and returns all clients except for the ones with matching connection ID.
- **Client**. Method which receives a string with a connection ID of a specific client and returns the matching client.
- **Clients**. Method which receives a list of string connection IDs of clients and returns all clients with a matching connection ID.
- **Group**. Method which receives a string group name and returns all clients which have been assigned to the matching group name.
- **GroupExcept**. Method which receives a string group name and a list of string connection IDs of clients and returns all clients which have been assigned to the matching group name with the exception of the matching connection IDs.
- **Groups**. Method which receives a list of string group names and returns all clients which have been assigned to the matching group name.
- **OthersInGroup**. Method which receives a string group name and returns all clients which have been assigned to the matching group name with the exception of the calling user.
- **User**. Method which receives a string user ID of a specific user, by default from the **ClaimTypes.NameIdentifier** and returns the matching user (This can apply to more than one client if the user is connected from multiple clients).
- **Users**. Method which receives a list of string user IDs, by default from the **ClaimTypes.NameIdentifier** and returns the matching user (This can apply to more than one client if the user is connected from multiple clients).

The hub can make multiple **SendAsync** requests, potentially with different method names, and perform calls to additional classes that perform additional logic. You can use dependency injection with hubs, as with other components of ASP.NET Core applications.

The following code is an example of a chat hub:

### Chat Hub

```
using Microsoft.AspNetCore.SignalR;

public class MyChatHub : Hub
{
 public async Task MessageAll(string sender, string message)
 {
 await Clients.All.SendAsync("NewMessage", sender, message);
 }
}
```

In this example, you can see a chat hub. Whenever a connected client invokes the **MessageAll** method, all clients will receive the message as well as the user that sent it. The clients will need to implement the **NewMessage** method in order to process it.

## SignalR Routing

Once the hub has been configured, you are able to map routes to it inside the **Configure** method. This can be done by calling the **MapHub** method on the **HubRouteBuilder** parameter. The **MapHub** method uses the following syntax: **MapHub<T>(\*path\*)**. **T** is the type of the hub which will be instantiated and **path** is a string which represents the relative URL path on which the SignalR hub will be hosted. Once this is done, the hub is hosted and accessible on the server.

The following code is an example of configuring SignalR hub routes:

### SignalR Hub Routing

```
app.UseSignalR(routes =>
{
 routes.MapHub<MyChatHub>("/mychathub");
});
```

In this example, we can see that the route **/mychathub** is mapped to the hub **MyChatHub** which was created earlier.

## SignalR in the Client

### Add SignalR Client Library

After the server is done, you will need to set up the client. Since controllers and views are rendered on the server during the initial request, you will need to use JavaScript to handle this. The first step is to add a dependency to the npm configuration. You will need to add a dependency to **@aspnet/signalr** as part of **package.json**.

The following code is an example of adding the SignalR client library to **package.json**:

### SignalR npm Package

```
{
 "version": "1.0.0",
 "name": "asp.net",
 "private": true,
 "dependencies": {
 "@aspnet/signalr": "1.0.0",
 "jquery": "3.3.1"
 },
 "devDependencies": {}
}
```

### Use SignalR Client JavaScript

Once this is done, you will need to build a view or HTML file to accommodate your logic and to interact with the SignalR. This file can be as simple or as complex as you wish and does not require any particular references to SignalR. However, you will need to load the SignalR JavaScript file from the npm package either in the layout or in the script itself. The SignalR JavaScript file will need to be called before your own scripts as with most third-party libraries.

The following code is an example of HTML code which will be used with SignalR:

### HTML Code

```
<div>
 <input type="text" class="username" />
 <input type="text" class="message" />
</div>
<div>
 <input type="button" class="all" value="Send To All" />
</div>
<div class="chatLog">
```

```
</div>

<script src="~/lib/jquery.js"></script>
<script src="~/lib/signalr.js"></script>
<script src="~/script/chat.js"></script>
```

In this example, you can see an HTML page with a text box for username, a text box for a message, a button for sending the message, and an empty div for holding future messages.

 **Note:** An important thing to note is that SignalR in ASP.NET Core MVC applications does not require jQuery to be used on the client side. It is only being added in order to interact with the DOM.

### Connect to a Hub

The next step is to set up the client to handle the SignalR connection. You will need to build a connection. In order to do this, you must create a new object of type **signalR.HubConnectionBuilder**. After this object is created, you will need to call the **withUrl(\*hubUrl\*)** method on the connection builder, specifying the URL of the hub, and finally you will need to call the **build** method, which returns the connection object that you will use.

The following code is an example of building the SignalR connection:

#### Building SignalR Connection

```
var connection = new signalR.HubConnectionBuilder()
 .withUrl("/mychathub")
 .build();
```

In this example, you can see that a new **signalR.HubConnectionBuilder** object is created. It is then provided with the relative URL **/mychathub**, and finally, the build method is called to create the connection.

### Register for Calls from the Hub

The next step in configuring the client is to register to calls from the server. This can be done by calling the **on(\*methodName\*, \*methodFunction\*)** method of the connection object. The method name will match the method called by **SendAsync** on the hub, while the method function will receive the parameters from the **SendAsync** call and process them as needed. You can add as many calls to the on method as needed to handle different methods from the hub. For example, in a chatroom there may be different methods declared for users leaving the chat, joining the chat, or sending messages.

The following code is an example of listening to a method call from the server:

#### SignalR Client Method

```
connection.on("NewMessage", (sender, message) => {
 $(".chatLog").text($(".chatLog").text() + " Message from " + sender + ":" + message);
});
```

In this example, you can see that the client implements the **NewMessage** method, which receives two parameters, **sender** and **message**. The **message** and **sender** will be appended to the chat log div as the string **Message from \*sender\*: \*message\***. This will occur every time the hub calls **NewMessage**.

## Start, Error Handling and Logging

Once all methods have been properly registered by using the **on** method, you will need to start the connection by calling the **start** method on the connection object. This method will start the connection with the server, by initially checking which transport types the server supports, before creating the actual connection, allowing the server and client to communicate through SignalR.

Since connections on the web are not infallible and errors occur, it is common to pipe a call to the **catch(\*errorFunction\*)** method after the call to **start**. This allows you to handle errors on the client side if something goes wrong. The **error** function receives a single parameter of the error that occurred. You can then use it to take any necessary actions.

The following code example demonstrates starting the SignalR connection:

### Starting the SignalR Connection

```
connection.start().catch(err => console.error(err.toString()));
```

In this example, you can see that the connection is started. If an error will occur, it will be printed in the browser console.

### Call Hub Methods from Client

Finally, to call methods on the hub, you will use the **invoke(\*methodName\*, \*param1\*, \*param2\*...)** method on the connection object. The method name is a string which must match a method that has been defined on the hub, while the remaining parameters should match the parameters of the method on the hub. Doing this will call the method which is defined on the hub.

The following code is an example of invoking a method on the hub:

### Invoke the Hub Method

```
$(".all").click(function () {
 var sender = $(".username").val();
 var message = $(".message").val();
 connection.invoke("MessageAll", sender, message);
});
```

In this example, you can see that when the user clicks the button with the **all** class, the values of the input with the **username** class and the input with the **message** class are extracted. They are then used when the connection object calls the **invoke** method, calling the **MessageAll** method on the hub and providing the **sender** and **message** as parameters.

 **Note:** It is important to note, SignalR connections which do not use WebSockets will require sticky servers to be defined. Since, by default, WebSockets create a socket with a specific server, you do not need any special logic to handle this case.

## Demonstration: How to Use SignalR

In this demonstration, you will first see how to configure SignalR in an ASP.NET Core application. After that you will see how to add a SignalR hub to the application. Then you will see how to connect to the hub from client-side code.

## Demonstration Steps

You will find the steps in the section "Demonstration: How to Use SignalR" on the following page:

[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD12\\_DEMO.md#demonstration-how-to-use-signalr](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD12_DEMO.md#demonstration-how-to-use-signalr).

## Additional SignalR Settings

Sometimes you need to call methods on specific clients. For instance, in a chatting website, there are usually multiple chat rooms, with users joining specific chat rooms and receiving messages only for that specific room. In addition, you will need to identify cases where users exit the chat room so they will no longer receive messages.

This can be managed by using the **Context** property of the hub class. The **Context** property is of the **HubCallerContext** type and can be used to identify the user that made the call.

Additionally, by overriding the **OnConnectedAsync** and **OnDisconnectedAsync** methods on the hub, you can perform initial and final operations on clients, allowing you to add or remove from groups or remove clients from being displayed.

Another useful property of the hub is the **Groups** property. This property is of the **IGroupManager** type and exposes the **AddToGroupAsync** and **RemoveFromGroupAsync** methods. Both methods receive a connection ID and a group to which the connection will be added to or removed from respectively. The connection ID can be found by using the **ConnectionId** property on the **Context** property. Note that if a client disconnects, the connection ID will be removed from all groups automatically.

- Hub members:
  - Properties: **Clients**, **Context**, **Groups**
  - Methods: **OnConnectedAsync**, **OnDisconnectedAsync**
- Serializing messages:
  - JSON
  - MessagePack
- Configuring connection:
  - Server-side: **HandshakeTimeout**, **KeepAliveInterval**, **SupportedProtocols**, **EnableDetailedErrors**
  - Client-side: **transport**, **serverTimeoutInMilliseconds**

 **Note:** The **Groups** property is exclusively used to add to and remove from **Groups** and nothing else. If you need a list of people inside a group or a list of groups you will need to manage it yourself.

The following code is an example of a hub for a chat room which uses groups:

### Chat Hub with Groups

```
public class MyChatHub : Hub
{
 private static Dictionary<string, string> _connectedUsers = new Dictionary<string, string>();

 public async Task MessageChatRoom(string message)
 {
 await Clients.Group("ChatRoom").SendAsync("NewMessage",
 _connectedUsers[Context.ConnectionId], message);
 }

 public async Task JoinChatRoom(string sender)
 {
 await Groups.AddToGroupAsync(Context.ConnectionId, "ChatRoom");
 _connectedUsers.Add(Context.ConnectionId, sender);
 }
}
```

```

 await Clients.Group("ChatRoom").SendAsync("UserListUpdate",
 _connectedUsers.Values);
}

public async Task LeaveChatRoom()
{
 await Groups.RemoveFromGroupAsync(Context.ConnectionId, "ChatRoom");
 _connectedUsers.Remove(Context.ConnectionId);
 await Clients.Group("ChatRoom").SendAsync("UserListUpdate",
 _connectedUsers.Values);
}

public override Task OnDisconnectedAsync(Exception exception)
{
 _connectedUsers.Remove(Context.ConnectionId);
 Clients.Group("ChatRoom").SendAsync("UserListUpdate", _connectedUsers.Values);
 return base.OnDisconnectedAsync(exception);
}
}

```

In this example, you can see a chat room being managed. Every user that joins is also added to the dictionary managed by the hub and whenever the active users changed, all logged in users receive an updated list of all connected users.

The following code is an example of the client-side code for the hub:

### Client Side for Groups

```

var connection = new signalR.HubConnectionBuilder()
 .withUrl("/mychathub")
 .build();

connection.on("NewMessage", (sender, message) => {
 $(".chatLog").text($(".chatLog").text() + " Message from " + sender + ":" + message);
});

connection.on("UserListUpdate", (userList) => {
 $(".users").text(userList);
});

connection.start().catch(err => console.error(err.toString()));

$(".join").click(function () {
 var sender = $(".username").val();
 connection.invoke("JoinChatRoom", sender);
});

$(".leave").click(function () {
 connection.invoke("LeaveChatRoom");
});

$(".message").click(function () {
 var message = $(".message").val();
 connection.invoke("MessageChatRoom", message);
});

```

In this example, you can see that the client invokes the methods declared in the hub and listens to the methods called by the hub.

### Serializing with MessagePack

By default, SignalR serializes message by using JSON. JSON, being a common message format, is easily serialized through different clients and is a widely recognized format. However, JSON is a text-based format and therefore less efficient at containing data than binary formats. While the performance change is negligible in small quantities of data, the size can add up in larger quantities and across many requests.

The MessagePack serialization format allows you to improve performance by sending smaller more compact data. As an additional benefit to security, it is much harder to read by looking at network traffic, as it requires the data to be converted from a MessagePack format, whereas JSON is immediately readable.

Using MessagePack requires a few extra steps to set up:

1. Install the **Microsoft.AspNetCore.SignalR.Protocols.MessagePack** Nuget package. This will load the required server-side package to use MessagePack.
2. In the **ConfigureServices** method of the **Startup** class, in the call to **AddSignalR** on the **IServiceCollection** object, pipe a call to **AddMessagePackProtocol**. This will cause the hub to serialize and deserialize in the MessagePack format.
3. Add a dependency for **@aspnet/signalr-protocol-msgpack** in the *package.json* file. This will retrieve the **signalr-protocol-msgpack** library, as well as **msgpack5**, which is a dependency for **signalr-protocol-msgpack**.
4. Add references to **msgpack5.js** and then to **signalr-protocol-msgpack.js** in the HTML files. The order of these two files is very important.
5. In the JavaScript file, as part of the **HubConnectionBuilder** pipe, add a call to **withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())** before the call to **build**. This will set up the connection to use message pack on the client.

The following code is an example of setting up MessagePack in **ConfigureServices**:

#### MessagePack in ConfigureServices

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddSignalR().AddMessagePackProtocol();
 services.AddMvc();
}
```

The following code is an example of loading MessagePack dependencies in HTML:

#### MessagePack Client Dependencies

```
<script src="~/lib/signalr.js"></script>
<script src="~/lib/msgpack5.js"></script>
<script src="~/lib/signalr-protocol-msgpack.js"></script>
```

The following code is an example of setting up MessagePack as part of the SignalR connection:

#### MessagePack on Client

```
var connection = new signalR.HubConnectionBuilder()
 .withUrl("/mychathub")
 .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
 .build();
```

 **Note:** It is also possible to use other format protocols. However, they will need additional third-party libraries or for you to write your own.

## Configuring the SignalR Connection

Sometimes there are additional requirements in running SignalR. For instance, you may not wish to maintain connections which are unused for some time or you may wish instead to keep up the connection as long as possible. Additionally, you may wish to remove support for specific transport protocols to improve performance or fulfill particular needs.

This can be done by using an overload of the **AddSignalR** method of the **ConfigureServices** class. The overload accepts an action, which gets an instance of the **HubOptions** class. In this class, you can set various configuration properties that will affect all **SignalR** connections on the application.

Common configurations include:

- **HandshakeTimeout**. This is the duration during which the client must send a message after the initial handshake or the connection will be closed.
- **KeepAliveInterval**. This is the duration during which the server will send a ping to the client to keep it alive. This is useful in a system where maintaining connections are important, but a long period could occur with no messages.
- **SupportedProtocols**. A collection which contains the protocols to be supported. Changing this collection can allow removing support for specific protocols.
- **EnableDetailedErrors**. A Boolean value which indicates if detailed error messages should be sent. By default, its value is **false**. Turning this to **true** will cause detailed exceptions to be sent. This could potentially present a security exploit.

The following code is an example of server-side SignalR configuration:

### Server-Side SignalR Configuration

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddSignalR(hubOptions =>
 {
 hubOptions.HandshakeTimeout = TimeSpan.FromSeconds(30);
 hubOptions.KeepAliveInterval = TimeSpan.FromSeconds(50);
 })
 .AddMessagePackProtocol();
 services.AddMvc();
}
```

In this example, you can see that the server is set to close a connection if no message has been sent within 30 seconds of the initial handshake. Additionally, once a connection has been established, the server will ping the client to prevent a timeout once every 50 seconds.

Additionally, it is also possible to set up configurations on the client side. It is possible to add an additional **options** parameter to the **withUrl** method, allowing you to set up additional configurations.

Common configuration options include:

- **transport**. Allows setting which transport types are used by the client. The client will not use any ones that are omitted and they are separated by a bitwise or operator.
- **serverTimeoutInMilliseconds**. Sets a period of time which the client will wait for a reply from the server, before closing the connection.

The following code is an example of client-side SignalR configuration:

### Client-Side SignalR Configuration

```
var connection = new signalR.HubConnectionBuilder()
 .withUrl("/mychathub", {
 transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.ServerSentEvents,
 serverTimeoutInMilliseconds: 20000
 })
 .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
 .build();
```

# Lab: Performance and Communication

## Scenario

You have been asked to create a web-based electric store application for your organization's customers. The application should have a page showing the sale of the day, products sorted by category, the ability to add products to your shopping list, and a chat page that lets users talk online. The application should contain a cache tag helper to cache content in a view, a memory cache in a controller, and a session state configuration. Finally, you will write a chat room app by using SignalR.

## Objectives

After completing this lab, you will be able to:

- Implement a caching strategy.
- Manage state.
- Add two-way communication.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD12\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD12_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD12\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD12_LAK.md).

## Exercise 1: Implementing a Caching Strategy

### Scenario

To improve the performance of a web application, caching should be used in the web application. In this exercise, you will first add a cache tag helper to a view. After that, you will use the memory cache to store and retrieve items.

The main tasks for this exercise are as follows:

1. Add a cache tag helper to a view
2. Insert data to be cached by the cache tag helper
3. Run the application
4. Insert items to a memory cache
5. Run the application

## Exercise 2: Managing State

### Scenario

To retain information across requests, the state should be used in the web application. In this exercise, you will use session state to manage state in the web application.

The main tasks for this exercise are as follows:

1. Enable working with sessions.
2. Use session to store values.
3. Retrieve values from a session.
4. Run the application and navigate from view to view.

## Exercise 3: Two-Way Communication

### Scenario

In this exercise, you will first add a SignalR Hub class named ChatHub. You will then add a SignalR Hub class named ChatHub, and register the ChatHub in the Startup class. Then, you will add a chat view. Finally, you will write the JavaScript code to connect to the server, and run the application and navigate from view to view.

The main tasks for this exercise are as follows:

1. Add a SignalR Hub class named ChatHub.
2. Register the ChatHub in the Startup class.
3. Add a chat view.
4. Write the JavaScript code to connect to the server.
5. Run the application.

**Question:** A member of your team added a product to the database. However, when he looks in the browser he can't see this product. Can you explain to him why?

**Question:** A member of your team changed the **Configure** method in the **Startup** class, so calling to the **UseMvc** middleware occurs before calling the **UseSignalR** middleware. Can you explain to him what is the impact of his change?

## Module Review and Takeaways

In this module, you have learned how to set up caching in your application, how to manage the application state, and how to use SignalR. By using caching, you have learned how to improve application performance by reducing logic being repeated. By using state, you have learned how to keep a persistent connection with the client and how to use it to change the behavior of the application. By using SignalR, you have learned how to create true two-way communication between client and server.

 **Note:** Remember that all of these technologies, while extremely helpful, should be considered before usage. If there is no complicated server-side logic or integration with external resources, caching might not be useful. If the application is designed to work as a stateless experience, there is no need to use state. And if the application does not require two-way communication, you should not use SignalR. As with any other feature, consider the benefits before using these.

### Review Question

**Question:** How would you use of the technologies covered in this topic on a shopping website?

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
After adding a SignalR hub a controller is not working	
SignalR connection appears to have timed out and is no longer working	

# Module 13

## Implementing Web APIs

### Contents:

Module Overview	13-1
<b>Lesson 1: Introducing Web APIs</b>	<b>13-2</b>
<b>Lesson 2: Developing a Web API</b>	<b>13-9</b>
<b>Lesson 3: Calling a Web API</b>	<b>13-21</b>
<b>Lab: Implementing Web APIs</b>	<b>13-30</b>
Module Review and Takeaways	13-32

## Module Overview

Most web applications require integration with external systems such as mobile applications. You need to know how to use Web APIs to promote application interaction with external systems. You can use the Web API to implement Representational State Transfer (REST) services in your application. REST services help reduce application overhead and limit the data that is transmitted between client and server systems. You need to know how to call a Web API by using server-side code and jQuery code to effectively implement REST-style Web APIs in your application.

### Objectives

After completing this module, you will be able to:

- Create services by using Microsoft ASP.NET Core Web API.
- Call a Web API from server-side code and jQuery.

# Lesson 1

## Introducing Web APIs

HTTP is a communication protocol that was created by Tim Berners-Lee and his team while working on the WorldWideWeb (later renamed to World Wide Web) project. Originally designed to transfer hypertext-based resources across computer networks, HTTP is an application layer protocol that acts as the primary protocol for many applications including the World Wide Web.

Because of its vast adoption and the common use of web technologies, HTTP is now one of the most popular protocols for building applications and services. In this lesson, you will be introduced to the basic structure of HTTP messages and understand the basic principles of the REST architectural approach.

Web API is a full-featured framework for developing HTTP-based services. Using Web API gives developers reliable methods for creating, testing, and deploying HTTP-based services. In this lesson, Web API will be introduced.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe what are HTTP services.
- Describe the structure of an HTTP message.
- Describe status codes.
- Describe Web APIs.

### HTTP Services

HTTP is a first-class application protocol that was built to power the World Wide Web. To support such a challenge, HTTP was built to allow applications to scale, taking into consideration concepts such as caching and stateless architecture. Today, HTTP is supported by many different devices and platforms reaching most computer systems available today.

HTTP also offers simplicity, by using text messages and following the request-response messaging pattern. HTTP differs from most application layer protocols because it was not designed as a Remote Procedure Calls (RPC) mechanism or a Remote Method Invocation (RMI) mechanism. Instead, HTTP provides semantics for retrieving and changing resources that can be accessed directly by using an address.

HTTP is used to develop both websites and services. Services developed by using HTTP are generally known as HTTP-based services.

### Using URI

Uniform Resource Identifier (URI) is an addressing standard that is used by many protocols. HTTP uses URI as part of its resource-based approach to identify resources over the network.

HTTP URIs follow this structure:

- HTTP is a first class application protocol
- An HTTP URI has the following basic structure:  


```

 http://blueyonder.com:8080/travelers?id=1
 +-----+ +-----+ +-----+ +-----+ +-----+
 | Schema | | Host | | Port | | Absolute Path | | Query |
 +-----+ +-----+ +-----+ +-----+ +-----+

```
- HTTP defines a set of methods or verbs that add action-like semantics to requests

"http://" host [ ":" port ] [ absolute path [ "?" query ]]

- **http://**. This prefix is standard to HTTP requests and defines the HTTP URI schema to be used.
- **Host**. The host component of the URI identifies a computer by an IP address or a registered name.
- **Port (optional)**. The port defines a specific port to be addressed. If not present, a default port will be used. Different schemas can define different default ports. The default port for HTTP is 80.
- **Absolute path (optional)**. The path provides additional data that together with the query describes a resource. The path can have a hierarchical structure similar to a directory structure, separated by the slash sign (/).
- **Query (optional)**. The query provides additional nonhierarchical data that together with the path describes a resource.

Different URIs can be used to describe different resources. For example, the following URIs describe different destinations in an airline booking system:

- <http://localhost/destinations/seattle>
- <http://localhost/destinations/london>

When accessing each URI, a different set of data, also known as a representation, will be retrieved.

## Using Verbs

HTTP defines a set of methods or verbs that add an action-like semantics to requests. Verbs are a central mechanism in HTTP and it is one of the mechanisms that make HTTP the powerful protocol it is. The following list shows the widely used verbs that are defined in HTTP:

- **GET**. Used to retrieve a representation of a resource. Requests intended to retrieve data based on the request URI.
- **POST**. Used to create, update, and by some protocols, retrieve entities from the server. Requests intended to send an entity to the server. The actual operation that is performed by the request is determined by the server. The server should return information about the outcome of the operation in the result.
- **PUT**. Used to create and update resources. Requests intended to store the entity sent in the request URI, completely overriding any existing entity in that URI.
- **DELETE**. Used to delete resources. Requests intended to delete the entity identified by the request URI.

## Introduction to REST

Representational State Transfer (REST) describes an architectural style that takes advantage of the resource-based nature of HTTP. It was first used in 2000 by Roy Fielding, one of the authors of the HTTP, URI, and HTML specifications. Fielding described, in his doctoral dissertation, an architectural style that uses some elements of HTTP and the World Wide Web for creating scalable and extendable applications.

Today, REST is used to add important capabilities to a service. Services that use the REST architectural style are also known as RESTful services. RESTful services use the different HTTP verbs to allow the user to manipulate the resources and create a full API based on resources.

## Media Types

HTTP was originally designed to transfer hypertext. Hypertext is a nonlinear format that contains references to other resources, some of which are other hypertext resources. However, some resources contain other formats such as image files and videos, which required HTTP to support the transfer of different types of message formats. To support different formats, HTTP uses Multipurpose Internet Mail Extensions (MIME) types, also known as media types.

In HTTP, media types are declared by using headers as part of a process that is known as content negotiation. When a client sends a request, it can send a list of requested media types, and in order of preference, it can accept them in the response. The server should try to fulfill the request for content as specified by the client. Content negotiation enables servers and clients to set the expectation of what content they should expect during their HTTP transaction.

## HTTP Messages

HTTP is a simple request-response protocol. All HTTP messages contain the following elements:

- Start-line
- Headers
- An empty line
- Body (optional)

Although requests and responses share the same basic structure, there are some differences between them of which you should be aware.

An HTTP request message:

```
GET http://localhost:4392/travelers/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, /*/*
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64;
Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:4392
DNT: 1
Connection: Keep-Alive
```

### Request Messages

Request messages are sent by the client to the server. Request messages have a specific structure based on the general structure of the HTTP messages.

This example shows an HTTP request message:

#### An HTTP Request

```
GET http://localhost:4392/travelers/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, /*/*
Accept-Language: en-US,en;q=0.7,he;q=0.3
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:4392
DNT: 1
Connection: Keep-Alive
```

The first and the most distinct difference between the request and response messages is the structure of the start-line, called request-lines.

#### Request-line

This HTTP request messages start-line has a typical request-line with the following space-delimited parts:

- **HTTP method.** This HTTP request message uses the GET method, which indicates that the client is trying to retrieve a resource.
- **Request URI.** This part represents the URI to which the message is being sent.
- **HTTP version.** This part indicates that the message uses HTTP version 1.1.

#### Headers

This request message also has several headers that provide metadata for the request. Although headers exist in both response and request messages, some headers are used exclusively by one of them. For example, the **Accept** header is used in requests to communicate the kinds of responses the clients would prefer to receive. This header is a part of a process known as *content negotiation*.

## Body

The request message has no body. This is typical of requests that use the GET method.

## Response Messages

Response messages also have a specific structure based on the general structure of HTTP messages.

This example shows an HTTP response message:

### The HTTP Response returned by the server for the above request

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/11.0.0.0
Date: Tue, 13 Nov 2012 18:05:11 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Content-Length: 188
Connection: Close

>{"TravelerId":1,"TravelerUserIdentity":"aaabbccc","FirstName":"FirstName1","LastName":"LastName1","MobilePhone":"555-555-5555","HomeAddress":"One microsoft road","Passport":"AB123456789"}
```

## Status-Line

HTTP response start-lines are called status-lines. This HTTP response message has a typical status-line with the following space-delimited parts:

- **HTTP version.** This part indicates that the message uses HTTP version 1.1.
- **Status-Code.** Status-codes help define the result of the request. This message returns a status-code of 200, which indicates a successful operation. Status codes will be covered in the next topic, "Status Codes".
- **Reason-Phrase.** A reason-phrase is a short text that describes the status code, providing a human-readable version of the status code.

## Headers

Like the request message, the response message also has headers. Some headers are unique for HTTP responses. For example, the **Server** header provides technical information about the server software being used. The **Cache-Control** and **Pragma** headers describe how caching mechanisms should treat the message.

Other headers, such as the **Content-Type** and **Content-Length**, provide metadata for the message body and are used in both requests and responses that have a body.

## Body

A response message returns a representation of a resource in JavaScript Object Notation (JSON). The JSON, in this case, contains information about a specific traveler in a travel management system. The format of the representation is communicated by using the **Content-Type** header describing the media type.

## Status Codes

Status-codes are three-digit integers returned as a part of response messages status-lines. Status codes describe the result of the effort of the server to satisfy the request. The next section of the status line after the status code is the reason-phrase, a human-readable textual description of the status-code.

Status codes are divided into five classes or categories. The first digit of the status code indicates the class of the status:

- Status codes describe the result of the server's attempt to process the request
- Status codes are constructed from a three-digit integer and a description called reason phrases
- HTTP has five different categories of status codes:
  - 1xx – Informational
  - 2xx – Success
  - 3xx – Redirection
  - 4xx – Client Error
  - 5xx – Server Error

Class	Usage	Examples
1xx – Informational	Codes that return informational response about the state of the connection.	<ul style="list-style-type: none"> <li>• 101 Switching Protocols</li> </ul>
2xx – Successful	Codes that indicate the request was successfully received and accepted by the server.	<ul style="list-style-type: none"> <li>• 200 OK</li> <li>• 201 Created</li> </ul>
3xx – Redirection	Codes that indicate that additional action should be taken by the client (usually in respect to different network addresses) in order to achieve the result that you want.	<ul style="list-style-type: none"> <li>• 301 Moved Permanently</li> <li>• 302 Found</li> <li>• 303 See Other</li> </ul>
4xx – Client Error	Codes that indicate an error that is caused by the client's request. This might be caused by a wrong address, bad message format, or any kind of invalid data passed in the client's request.	<ul style="list-style-type: none"> <li>• 400 Bad Request</li> <li>• 401 Unauthorized</li> <li>• 404 Not Found</li> </ul>
5xx – Server Error	Codes that indicate an error that was caused by the server while it tried to process a seemingly valid request.	<ul style="list-style-type: none"> <li>• 500 Internal Server</li> <li>• 505 HTTP Version Not Supported</li> </ul>

## Introduction to Web API

HTTP has been around ever since the World Wide Web was created in the early 1990s, but its adoption as an application protocol for developing services took time. In the early years of the web, SOAP was considered the application protocol of choice by most developers. SOAP provided a robust platform for developing RPC-style services.

With the appearance of internet-scale applications and the growing popularity of web 2.0, it became clear that SOAP was not fit for such challenges and HTTP received more and more attention.

- For a long time the .NET Framework did not have a first-class framework for building HTTP services
- The need for developing HTTP services justified creating a new framework
- In February 2012, ASP.NET Web API was released
- In June 2016, ASP.NET Core Web API was released

### HTTP in the .NET Framework and the .NET Core

For the better part of the first decade of its existence, the .NET Framework did not have a first-class framework for building HTTP services. At first, ASP.NET provided a platform for creating HTML-based web-pages and ASP.NET web services, and later-on Windows Communication Foundation (WCF) provided SOAP-based platforms. For these reasons, HTTP never received the attention it deserved.

When WCF first came out in .NET 3.0, it was a SOAP-only framework. As the world started to use HTTP as the application-layer protocol for developing services, Microsoft started to make investments in extending WCF for supporting simple HTTP scenarios. By the next release of the .NET Framework (.NET 3.5), WCF had new capabilities. These included a new kind of binding called **WebHttpBinding** and the new attributes for mapping methods to HTTP requests.

In 2009, Microsoft released the WCF REST Starter Kit. This added the new **WebServiceHost** class for hosting HTTP-based services, and also new capabilities like help pages and Atom support. When the .NET Framework version 4 was released most of the capabilities of the WCF REST Starter Kit were already rolled into WCF. This includes support for IIS hosting in addition to new Visual Studio templates available through the Visual Studio Extensions Manager. But even then, WCF was still missing support for many HTTP scenarios.

The need for a comprehensive solution for developing HTTP services in the .NET Framework justified creating a new framework. Therefore, in October 2010, Microsoft announced the WCF Web API, which introduces a new model and additional capabilities for developing HTTP-based services. These capabilities included:

- Better support for content negotiation and media types.
- APIs to control every aspect of the HTTP messages.
- Testability.
- Integration with other relevant frameworks like Entity Framework and Unity.

The WCF Web API team released six preview versions until in February 2012, they were united with the ASP.NET team, forming the ASP.NET Web API.

In June 2016, Microsoft released the first version of .NET Core and ASP.NET Core. ASP.NET Core Web API is the framework for developing HTTP services in .NET Core.

## What is a Web API?

Web API is a framework that enables you to build REST-enabled APIs. REST-enabled APIs help external systems use the business logic implemented in your application to increase the reusability of the application logic. Web API facilitates two-way communication between the client system and the server through tasks such as:

- Instructing an application to perform a specific task
- Reading data values
- Updating data values

### Web API:

- Helps create REST-style APIs
- Enables external systems to use the business logic implemented in your application
- Uses URLs in requests and helps obtain results
- Is ideal for mobile application integration



Web API enables developers to obtain business information by using REST, without creating complicated XML requests such as Simple Object Access Protocol (SOAP). Web APIs use URLs in requests, thereby eliminating the need for complicated requests. For example, the following URL obtains information for a customer entity with the ID **1**:

<http://api.contoso.com/api/customers/1>

Web API uses such URLs in requests and obtains results in the JSON format by default. The following code shows a Web API response in the JSON format:

### A Web API JSON Response

```
[{"Id":1,"Name":"Tomato soup","Category":"Groceries","Price":1.0},{ "Id":2,"Name":"Yo-yo","Category":"Toys","Price":3.75},{ "Id":3,"Name":"Hammer","Category":"Hardware","Price":16.99}]
```

REST and Web API enable all kinds of different applications, including mobile device applications, to interact with services. In particular, REST and Web API provide the following benefits for mobile applications:

- They reduce the processing power needed to create complex request messages for data retrieval.
- They enhance the performance of the application by reducing the amount of data exchange between client and server.

## Lesson 2

# Developing a Web API

You need to know how to develop Web API for applications because Web API facilitates creating APIs for mobile applications, desktop applications, web services, web applications, and other applications. By creating a Web API, you make the information in your web application available for other developers to use in their systems. Each web application has a different functional methodology; this difference can cause interoperability issues in applications. REST services have a lightweight design, and Web API helps implement REST services to solve the interoperability issues. You need to know how to use the different routing methods that ASP.NET Core provides to implement REST services.

### Lesson Objectives

After completing this lesson, you will be able to:

- Create a Web API for an ASP.NET Core web application.
- Describe REST services.
- Explain how to use routes and controllers to implement REST in Web APIs.
- Pass parameters to a Web API action.
- Control the response returned from a Web API action.
- Describe data return formats.
- Explore a Web API by using Microsoft Edge as a client.

### Using Routes and Controllers

Web API uses controllers and actions to handle requests. A controller is a class that derives from the **ControllerBase** base class. By convention, controllers are named with the **Controller** suffix. Controllers contain methods, known as actions, which process the requests and return the results.

The following code shows a controller named **HomeController** with an action named **Get**:

#### A Controller with an Action

```
public class HomeController : ControllerBase
{
 public string Get()
 {
 return "Response from Web API";
 }
}
```

#### Obtaining information by using Web API:

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 public string Get()
 {
 return "Response from Web API";
 }
}
```

### Defining Routes

Web API uses routing rules to map HTTP requests to the Web API controllers and actions by using HTTP verbs and the request URL. You can configure routes by using convention-based routing and you can configure routes by using attributes.



**Note:** Routes are covered in Module 4, "Developing Controllers".



**Best Practice:** Although it is possible to configure routes of Web API by using convention-based routing, the recommendation is to prefer configuring the routes of Web API by using attributes.

The following code shows how to configure a route by using an attribute:

### Configuring a Route

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 public string Get()
 {
 return "Response from Web API";
 }
}
```

In the preceding code sample, observe that the route includes the literal path segment **api**. This segment ensures that Web API requests are clearly separate from MVC controller routes. The placeholder variable, **[controller]** helps identify the API controller to which to forward the request. As for MVC controllers, Web API appends the word, Controller, to this value to locate the right API controller class. For example, Web API routes a request to the **api/Home** URI to the **HomeController** controller.

## RESTful Services

REST uses URLs and HTTP verbs to uniquely identify the entity that it operates on and the action that it performs. REST helps retrieve business information from the server. However, in addition to data retrieval, business applications perform more tasks such as creating, updating, and deleting information on the database. Web API and REST facilitate handling such additional tasks. They use the HTTP method to identify the operation that the application needs to perform.

The following table provides information on some HTTP methods that Web API and REST use.

#### Characteristics of a REST Service:

- Can be called to retrieve business information from the server
- Can create, update, and delete information in a database through HTTP operations
- Uses URLs to uniquely identify the entity that it operates on
- Uses HTTP verbs to identify the operation that the application needs to perform. The HTTP verbs include:
  - **GET**
  - **POST**
  - **PUT**
  - **DELETE**

HTTP Verb	Description
<b>GET</b>	Use this method with the following URL to obtain a list of all customers. <b>/api/customers</b>
<b>GET</b>	Use this method with the following URL to obtain a customer by using the ID detail. <b>/api/customers/id</b>

HTTP Verb	Description
<b>GET</b>	Use this method with the following URL to obtain customers by using the category detail. <code>/api/customers?country=country</code>
<b>POST</b>	Use this method with the following URL to create a customer record. <code>/api/customers</code>
<b>PUT</b>	Use this method with the following URL to update a customer record. <code>/api/customers/id</code>
<b>DELETE</b>	Use this method with the following URL to delete a customer record. <code>/api/customers/id</code>

Web API allows developers to use a strong typed model for developers to manipulate HTTP requests. The following code shows how to use the POST, PUT, and DELETE methods for the create, update, and delete requests to handle the creation, retrieval, updating, and deletion (CRUD) of the customer records:

### CRUD Operations

```
[Route("api/[controller]")]
public class CustomerController : ControllerBase
{
 public IEnumerable<Customer> Get()
 {
 // Fill content here
 }

 public void Post(Customer item)
 {
 // Fill content here
 }

 public void Put(int id, Customer item)
 {
 // Fill content here
 }

 public void Delete(int id)
 {
 // Fill content here
 }
}
```

## Action Methods and HTTP Verbs

When a client requests a Web API, first the controller is chosen. The next step is choosing the method that will handle the request. The method selection can be done based on the request that the HTTP method has used and on the request-URI.

### The **HttpGet**, **HttpPut**, **HttpPost**, and **HttpDelete** Attributes

You can use the **HttpGet**, **HttpPut**, **HttpPost**, or **HttpDelete** attributes in your controller action to specify that a function is mapped to a specific HTTP verb. The following table describes how the HTTP attributes map to the HTTP verbs.

You can use the following attributes to control the mapping of HTTP requests (HTTP verb+URL) to actions in the controller:

- The **HttpGet**, **HttpPut**, **HttpPost**, or **HttpDelete** attributes
- The **AcceptVerbs** attribute
- The **ActionName** attribute

Attribute	HTTP Verb
HttpGet	GET
HttpPut	PUT
HttpPost	POST
HttpDelete	DELETE

The following code illustrates the use of the **HttpGet** attribute on the **SomeMethod** action:

### Using the **HttpGet** Attribute

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [HttpGet]
 public string SomeMethod()
 {
 return "SomeMethod was invoked";
 }

 public string OtherMethod()
 {
 return "OtherMethod was invoked";
 }
}
```

In the preceding code sample, Web API routes a request to the **api/Home** URI to the **SomeMethod** action.

 **Note:** In the preceding code example, notice that if you delete the **HttpGet** attribute, an exception will be thrown when there is a request to the **api/Home** route. The reason is that in this case both **SomeMethod** and **OtherMethod** match the route data and it is impossible to decide which one should be invoked.

If you want to have multiple methods that are mapped to the same HTTP verb, you can specify a **template** parameter to the **Http[verb]** attribute.

The following code example demonstrates how to specify a **template** parameter to the **HttpGet** attribute:

### Passing a template Parameter to the **HttpGet** Attribute

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [HttpGet("Some")]
 public string SomeMethod()
 {
 return "SomeMethod was invoked";
 }

 [HttpGet("Other")]
 public string OtherMethod()
 {
 return "OtherMethod was invoked";
 }
}
```

In the preceding code sample, Web API routes a request to the URI, **api/Home/Some**, to the action called **SomeMethod**. Web API routes a request to the URI, **api/Home/Other**, to the action called **OtherMethod**.

### The **AcceptVerbs** Attribute

The use of the **AcceptVerbs** attribute allows you to specify multiple HTTP verbs to the same actions in the controller.

The following code shows the use of the **AcceptVerbs** attribute to map specific HTTP verbs to the action:

### Using the **AcceptVerbs** Attribute

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [AcceptVerbs("Get", "Head")]
 public string SomeMethod()
 {
 return "SomeMethod was invoked";
 }

 public string OtherMethod()
 {
 return "OtherMethod was invoked";
 }
}
```

In the preceding code sample, Web API routes a request to the **api/Home** URI to the **SomeMethod** action.

### The **ActionName** attribute

In addition to the **controller** placeholder in routes, ASP.NET Core Web API also has a special placeholder for action.

In the following code sample, Web API routes a request to the **api/Home/SomeMethod** URI to the **SomeMethod** action:

## Using a Placeholder for an Action

```
[Route("api/[controller]/[action]")]
public class HomeController : ControllerBase
{
 [HttpGet]
 public string SomeMethod()
 {
 return "SomeMethod was invoked";
 }

 public string OtherMethod()
 {
 return "OtherMethod was invoked";
 }
}
```

You can use the **ActionName** attribute to specify the action name to be used in the routing.

The following code shows how to map an action to an HTTP request by using a custom action name:

## Using the ActionName Attribute

```
[Route("api/[controller]/[action]")]
public class HomeController : ControllerBase
{
 [HttpGet]
 [ActionName("SomeAction")]
 public string SomeMethod()
 {
 return "SomeMethod was invoked";
 }

 public string OtherMethod()
 {
 return "OtherMethod was invoked";
 }
}
```

In the preceding code sample, Web API routes a request to the **api/Home/SomeAction** URI to the **SomeMethod** action.

## Binding Parameters to Request Message

After locating the controller and action method, mapping data from the HTTP request to method parameters should be done. In ASP.NET Core Web API, this process is known as parameter-binding.

HTTP messages data can be passed in the following ways:

- **The message-URI.** In HTTP, the absolute path and query are used to pass simple values that help identify the resource and influence the representation.
- **The Entity-body.** In some HTTP messages, the message body passes data.

- An action that gets two parameters:

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [HttpGet("{id}/{name}")]
 public string Get(int id, string name)
 {
 return "id: " + id + ", name: " + name;
 }
}
```

- This action method is chosen when sending a GET request by using the **api/Home/1/Mike** path

By default, ASP.NET Core Web API differentiates simple and complex types. Simple types are mapped from the URI and complex types are mapped from the entity-body of the request.

The following code shows an action that gets the **id** parameter. The route is configured so the **id** parameter is mapped as part of the absolute path of the URI. This action method is chosen when sending a GET request by using the **api/Home/1** path:

### An Action that Gets a Parameter

```
[Route("api/[controller]/'{id}')"]
public class HomeController : ControllerBase
{
 [HttpGet]
 public string GetId(int id)
 {
 return "Input number is: " + id;
 }
}
```

You can use the **template** parameter of the **HttpGet**, **HttpPut**, **HttpPost**, and **HttpDelete** attributes to map the route and the parameters of the action.

The following code shows how to pass a template to the **HttpGet** attribute. The route is configured so the **id** parameter is mapped as part of the absolute path of the URI. This action method is chosen when sending a GET request by using the **api/Home/1** path:

### Passing a Template to **HttpGet**

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [HttpGet("{id}")]
 public string GetId(int id)
 {
 return "Input number is: " + id;
 }
}
```

The following code shows an action that gets two parameters named **id** and **name**. The route is configured so the **id** and **name** parameters are mapped as part of the absolute path of the URI. This action method is chosen when sending a GET request by using the **api/Home/1/Mike** path:

### An Action that Gets Two Parameters

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
 [HttpGet("{id}/{name}")]
 public string Get(int id, string name)
 {
 return "id: " + id + ", name: " + name;
 }
}
```

### Binding Source Attributes

A parameter's value of an action can be found in several locations. The location can be defined in the binding source attributes. There are several binding source attributes which include:

- **FromBody**. Determines that the parameter is bound by using the request body.
- **FromQuery**. Determines that the parameter is bound by using the request query string.
- **FromRoute**. Determines that the parameter is bound by using route-data from the request.

- **FromForm**. Determines that the parameter is bound by using form-data in the request body.
- **FromHeader**. Determines that the parameter is bound by using the request headers.

The following example demonstrates how to use the **FromBody** attribute to determine that a parameter is bound by using the request body:

### Using the FromBody Attribute

```
[Route("api/[controller]")]
public class CustomerController : ControllerBase
{
 [HttpPost]
 public void Post([FromBody] Customer item)
 {
 }
}
```

### Using the ApiController Attribute

It is possible to annotate a controller with the **ApiController** attribute. Annotating a controller with the **ApiController** attribute indicates that the controller serves HTTP API requests. It is used to target conventions on the controller. An example of a convention is the inference of binding source parameters.

In case you have a controller which is not annotated with the **ApiController** attribute, the binding source attributes should be defined explicitly. In a controller which is annotated with the **ApiController** attribute, there are inference rules which determine the data sources of the action parameters. For example, for complex type parameters, **FromBody** is inferred.

The following code example demonstrates how to use the **ApiController** attribute to infer **FromBody** for the parameter:

### ApiController Example

```
[Route("api/[controller]")]
[ApiController]
public class CustomerController : ControllerBase
{
 [HttpPost]
 public void Post(Customer item)
 {
 }
}
```

## Control the HTTP Response

Action methods can return both simple and complex types that are serialized to a data return format based on the content negotiation.

Although ASP.NET Core Web API can handle the content negotiation and serialization, it is sometimes required to handle other aspects of the HTTP response message (for example, returning a status code other than 200).

When an action method returns an object, ASP.NET Core Web API automatically serializes the object and returns it. The status code of the response in this case is 200 (OK) in case no

HTTP responses use status codes to express the outcome of the request processing

```
public IActionResult Get(string id)
{
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return Ok(_items[id]);
}
```

exception was thrown. In case there is an unhandled exception in the action method, a 5xx error is returned.

The following code shows an action method that returns a value of the **string** type. In case the **id** parameter is not negative, the status code of the response is 200. In case the **id** parameter is negative, the status code of the response is 500:

### Status Code Example

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 [HttpGet("{id}")]
 public string Get(int id)
 {
 if (id >= 0)
 return "value";
 throw new ArgumentException();
 }
}
```

### Return IActionResult

An action method can return an object that implements the **IActionResult** interface. This gives the flexibility to return a status code that is different from 200. For example, to return a 404 Not Found status code you can return a **NotFoundResult** object by using the **NotFound** method. If the status code of the response is 200, you can use an **Ok** method to return the content.

The following example shows an action method named **Get** that gets a parameter named **id**. If a dictionary contains an item with a key that is equal to **id** then the value is returned by using the **Ok** method, otherwise a response with status code 404 is returned:

### Return IActionResult from an Action Method

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public IActionResult Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return Ok(_items[id]);
 }
}
```

### Return ActionResult<T>

In addition to returning an **IActionResult**, a Web API controller action can return **ActionResult<T>**. Returning **ActionResult<T>** enables you to return a specific type or an object which inherits from **ActionResult**.

The following example demonstrates how an action can return **ActionResult<T>**. This example is equivalent to the previous example in which the action returned **IActionResult**:

#### Return ActionResult<T> from an Action Method

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public ActionResult<string> Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return _items[id];
 }
}
```

## Data Return Formats

When a client makes a request to a Web API controller, the controller action often returns some data. Web API can return this data in different formats, including JSON and XML. Both JSON and XML are text formats that represent information as strings.

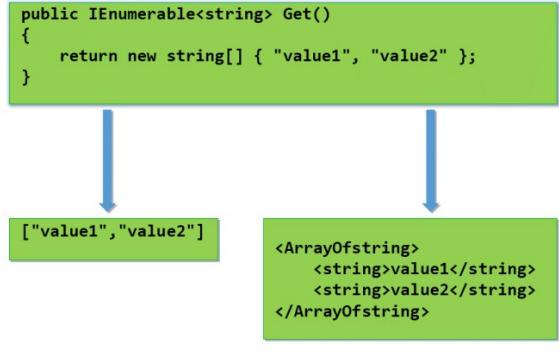
When a client makes a request, the client can specify the data format for the response. If the data format is not specified, Web API formats data as JSON by default.

The following code shows a controller named **PersonController** controller with the **Get** action:

#### A Controller with an Action

```
public class Person
{
 public int ID { get; set; }
 public string Name { get; set; }
}

[Route("api/[controller]")]
[ApiController]
public class PersonController : ControllerBase
{
 public IEnumerable<Person> Get()
 {
 List<Person> people = new List<Person>();
 people.Add(new Person() { ID = 1, Name = "Bob" });
 people.Add(new Person() { ID = 2, Name = "Mike" });
 }
}
```



```

 return people;
 }
}

```

The following code shows a JSON response from the **Get** action:

### A JSON Response

```
[{"ID":1,"Name":"Bob"}, {"ID":2,"Name":"Mike"}]
```

## Output Formatters

Web API uses an output formatter to format or serialize the information that a Web API REST service returns. Web API usually uses the default formatter to return a data in JSON format from an action. However, you can alternatively use other output formatters. To use an output formatter, you need to configure ASP.NET Core MVC to support it. You can use pre-defined output formatters such as an XML output formatter. You can also use custom output formatters to format the responses.

### Return Data in XML Format

XML output formatters are located in the **Microsoft.AspNetCore.Mvc.Formatters.Xml** package. Therefore, in case you would like your action to return data in the XML format, install the **Microsoft.AspNetCore.Mvc.Formatters.Xml** package.

To use an XML output formatter, you can add the formatter to an **MvcOptions** object and pass it as a parameter to the **AddMvc** method inside the **ConfigurationServices** method of the **Startup** class.

The following example shows how to add the **XmlSerializerOutputFormatter**:

#### Adding XmlSerializerOutputFormatter

```

public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc(options =>
 {
 options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
 });
}

```

Alternately, you can add the XML serializer formatters to MVC by using the **AddXmlSerializerFormatters** method, as shown in the following example:

#### Using the AddXmlSerializerFormatters Method

```

public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc().AddXmlSerializerFormatters();
}

```

Both approaches will serialize the same results.

The following code shows an XML response from the **Get** action of the **PersonController**: controller:

### An XML Response

```

<ArrayOfPerson>
 <Person>
 <ID>1</ID>
 <Name>Bob</Name>
 </Person>
 <Person>
 <ID>2</ID>
 <Name>Mike</Name>
 </Person>

```

```
</ArrayOfPerson>
```

## Specifying the Return Format

You can use the **ProducesAttribute** attribute to force a particular format of the response from an action.

The following example shows how to set the **Get** action to return data in the XML format:

### Using the ProducesAttribute Attribute

```
[Route("api/[controller]")]
[ApiController]
public class PersonController : ControllerBase
{
 [Produces("application/xml")]
 public IEnumerable<Person> Get()
 {
 List<Person> people = new List<Person>();
 people.Add(new Person() { ID = 1, Name = "Bob" });
 people.Add(new Person() { ID = 2, Name = "Mike" });
 return people;
 }
}
```

## Demonstration: How to Develop a Web API

In this demonstration, you will learn how to create a Web API controller. After this, you will see how to add actions to the controller. Then you will see how to call the actions by using Microsoft Edge. Finally, you will see how to control the data return format.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Develop a Web API” on the following page:  
[https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD13\\_DEMO.md#demonstration-how-to-develop-a-web-api](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD13_DEMO.md#demonstration-how-to-develop-a-web-api).

## Lesson 3

# Calling a Web API

After you complete the development of the Web API, you can create the client applications that call it. You need to know how to call a Web API by using jQuery code. You also need to know how to call a Web API by using server-side code, how to get a complex object from a Web API, and how to pass a complex object to a Web API.

### Lesson Objectives

After completing this lesson, you will be able to:

- Call Web APIs by using jQuery.
- Call Web APIs by using server-side code.
- Get complex objects from a Web API.
- Pass complex objects to a Web API.

### Calling Web APIs by Using jQuery Code

When HTTP was built in the early 1990s, it was made for a very specific kind of client: web browsers running HTML. Before the creation of JavaScript in 1995, HTML was using two of the three HTTP methods in HTTP 1.0: GET and POST. GET requests are usually invoked by entering a URI in the address bar or in kinds of hypertext references such as **img** and **script** tags.

Another way to start HTTP requests from a browser is by using HTML forms. HTML forms are HTML elements that create a form-like UI in the HTML document that lets the user insert and submit data to the server. HTML forms contain sub-elements, called input elements, and each represents a piece of data both in the UI and in the resulting HTTP message.

The most flexible mechanism to start HTTP from a browser environment is by using JavaScript. Using JavaScript provides two main capabilities that are lacking in other browser-based techniques:

- Complete control over the HTTP requests (including HTTP method, headers, and body).
- Asynchronous JavaScript and XML (AJAX). Using AJAX, you can send requests from the client after the browser completes loading the HTML. Based on the result of the calls, you can use JavaScript to update parts of the HTML page.

- You can use jQuery to generate an HTTP request from a browser to a Web API by using the **jQuery ajax** function
- You can use **JSON.stringify** in the **data** parameter of the **ajax** function to serialize the JavaScript objects into JSON objects

### Calling Web API Get Method by Using jQuery

jQuery is a JavaScript library that simplifies JavaScript programming. Among other things, jQuery simplifies calling to a Web API from a browser.

 **Note:** jQuery is introduced in Module 8, "Using Layouts, CSS and JavaScript in ASP.NET Core MVC".

You can use jQuery to generate an HTTP request from a browser to a Web API by using the jQuery **ajax** function. In the following example, you will first see a Web API controller that has a **Get** method and then you will see how to call the **Get** method by using the jQuery **ajax** function.

The following code shows a Web API controller called **ValuesController** that has a **Get** method:

### A Controller with Get Method

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public ActionResult<string> Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return _items[id];
 }
}
```

In the example above, the **Get** method can be invoked by using the relative URL **api/Values/{id}**. The **id** segment should be specified by the client. For example, if a client requests the relative URL **api/Values/key1**, the content **value1** will be returned to the client and the status code of the response will be 200. On the other hand, in case a client requests the relative URL **api/Values/key3**, the status code of the response will be 404 (Not Found).

The following code shows how to use jQuery to call the **Get** method of the **ValuesController** Web API controller by using the relative URL **api/Values/key1**. Running this code causes the text **value1** to be displayed on the browser:

### Calling the Get Method

```
<!DOCTYPE html>
<html>
<head>
 <title>Index</title>
 <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.js">
 </script>
 <script>
 $(function() {
 $.ajax({
 url: "http://localhost:[port]/api/values/key1",
 type: "GET"
 }).done(function (responseText) {
 $("#myDiv").text(responseText);
 }).fail(function () {
 alert("An error has occurred");
 });
 });
 </script>
</head>
<body>
 <div id="myDiv"></div>
</body>
```

```
</html>
```

If the **url** parameter is changed to **http://localhost:[port]/api/values/key3**, running the code will cause a pop-up window with the text **An error has occurred** to be displayed in the browser.

## Calling Web API Post Method by Using jQuery

It is possible to use the **ajax** function to call other methods in a Web API controller, such as a **Post** method. In the following example, you will first see a Web API controller that has a **Post** method and then you will see how to call the **Post** method by using the jQuery **ajax** function.

The following code shows a Web API controller called **ValuesController** that has a **Post** method. It receives an instance of type **Entry** that has a key and a value:

### A Controller with a Post Method

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public ActionResult<string> Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return _items[id];
 }

 [HttpPost]
 public IActionResult Post(Entry entry)
 {
 if (_items.ContainsKey(entry.Key) == true)
 {
 return BadRequest();
 }

 _items.Add(entry.Key, entry.Value);
 return CreatedAtAction(nameof(Get), new { id = entry.Key }, entry);
 }
}

public class Entry
{
 public string Key { get; set; }
 public string Value { get; set; }
}
```

In the **Post** method, the **BadRequest** and **CreatedAtAction** methods are used to give the client an indication of whether the **Post** method succeeded or failed. In case the **Post** method succeeds, a status code of 201 (Created) is returned to the client. In case the **Post** method fails, a status code of 400 (Bad Request) is returned to the client.

The following code shows how to use jQuery to call the **Post** method of the **ValuesController** Web API controller:

### Calling the Post Method

```
<!DOCTYPE html>
<html>
<head>
 <title>Index</title>
 <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.js">
 </script>
 <script>
 $(function() {
 var params = { key: 'key3', value: 'value3' };
 $.ajax({
 url: "http://localhost:[port]/api/values",
 type: "POST",
 data: JSON.stringify({ key: 'key3', value: 'value3' }),
 contentType: "application/json; charset=utf-8"
 }).done(function (responseText) {
 $("#myDiv").text("Value added successfully");
 }).fail(function () {
 alert("An error has occurred");
 });
 });
 </script>
</head>
<body>
 <div id="myDiv"></div>
</body>
</html>
```

In the preceding code sample, observe the **data** parameter of the **ajax** function. You can use the **JSON.stringify** method in the **data** parameter of the **ajax** function to serialize the JavaScript object into a JSON object, which will be sent to the Web API method.

## Demonstration: How to call Web APIs by Using jQuery code

In this demonstration, you will first see how to add a Web API controller with Get and Post methods. Then you will see how to call the Web API methods by using jQuery.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Call Web APIs by Using jQuery Code” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD13\\_DEMO.md#demonstration-how-to-call-web-apis-using-jquery-code](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD13_DEMO.md#demonstration-how-to-call-web-apis-using-jquery-code).

## Calling Web APIs by using Server-Side Code

ASP.NET Core provides a client-side API for consuming Web APIs. The main class for this API is **System.Net.Http.HttpClient**. This provides basic functionality for sending requests and receiving responses.

**HttpClient** keeps a consistent API with ASP.NET Core web API by using **HttpRequestMessage** and **HttpResponseMessage** for handling HTTP messages. The **HttpClient** API is a task-based asynchronous API providing a simple model for consuming HTTP asynchronously.

In order to consume ASP.NET Core Web API methods, the **HttpClient** class provides several methods, which include:

- **GetAsync**. Sends a GET request to a specified URI.
- **PostAsync**. Sends a POST request to a specified URI.
- **PutAsync**. Sends a PUT request to a specified URI.
- **DeleteAsync**. Sends a DELETE request to a specified URI.

To create **HttpClient** instances, you can use the **IHttpClientFactory** service. To register the **IHttpClientFactory** service you can call the **AddHttpClient** method inside of the **ConfigureServices** method.

The following code demonstrates how to register the **IHttpClientFactory** service:

### Register the **IHttpClientFactory** Service

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddMvc();
 services.AddHttpClient();
}
```

After the **IHttpClientFactory** service is registered, dependency injection can be used to inject it. Once injected, you can use the **CreateClient** method to get an instance of type **HttpClient**.

The following example demonstrates how to create an instance of type **HttpClient** inside a controller:

### Creating an **HttpClient** Instance

```
public class HomeController : Controller
{
 private IHttpClientFactory _httpClientFactory;

 public HomeController(IHttpClientFactory httpClientFactory)
 {
 _httpClientFactory = httpClientFactory;
 }

 public IActionResult Index()
 {
 HttpClient httpClient = _httpClientFactory.CreateClient();
 return View();
 }
}
```

To call Web APIs by using server-side code:

- Add code to initialize the **HttpClient** class
- Add code to create requests by using **GetAsync**, **PostAsync**, **PutAsync** and **DeleteAsync**

```
HttpClient client = _httpClientFactory.CreateClient();
client.BaseAddress = new Uri("http://localhost:[port]");
 HttpResponseMessage response =
 client.GetAsync("api/values/key1").Result;
```

## Calling Web API Get Method by Using HttpClient

In the following example, you will first see a Web API controller that has a **Get** method and then you will see how to call the **Get** method by using the **GetAsync** method of the **HttpClient** class.

The following code shows a Web API controller called **ValuesController** that has a **Get** method:

### A Web API Controller with a Get Method

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public ActionResult<string> Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return _items[id];
 }
}
```

The following code shows how to use the **HttpClient** class to call the **Get** method of the **ValuesController** Web API controller by using the relative URL **api/Values/key1**. Running this code causes the text **value1** to be displayed on the browser:

### Calling the Get Method

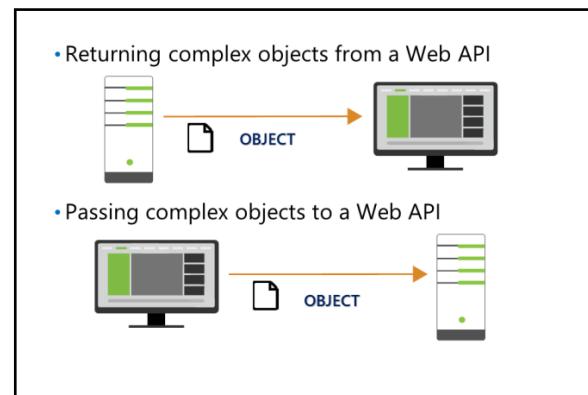
```
public class HomeController : Controller
{
 private IHttpClientFactory _httpClientFactory;

 public HomeController(IHttpClientFactory httpClientFactory)
 {
 _httpClientFactory = httpClientFactory;
 }

 public IActionResult Index()
 {
 HttpClient httpClient = _httpClientFactory.CreateClient();
 httpClient.BaseAddress = new Uri("http://localhost:[port]");
 HttpResponseMessage response = httpClient.GetAsync("api/values/key1").Result;
 if (response.IsSuccessStatusCode)
 {
 string result = response.Content.ReadAsStringAsync().Result;
 return Content(result);
 }
 else
 {
 return Content("An error has occurred");
 }
 }
}
```

## Working with Complex Objects

In the previous topic, you saw how a client can require string representation of the data. When working with complex objects, a more useful approach is to obtain a deserialized object based on the entity body. In the following example, you will first see a Web API controller that has a **Get** method that returns an instance of type **Person**. Then you will see how to call the **Get** method by using the **GetAsync** method of the **HttpClient** class and use the **ReadAsAsync** method to deserialize the response to an instance of type **Person**.



The following code shows the **Person** class:

### The Person Class

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

The following code shows a Web API controller called **PersonController** that has a **Get** method. The **Get** method returns an instance of type **Person**:

### A Web API that Returns an Object

```
[Route("api/[controller]")]
[ApiController]
public class PersonController : ControllerBase
{
 public Person Get()
 {
 return new Person() { Id = 1, Name = "Mike" };
 }
}
```

The following code shows how to use the **HttpClient** class to call the **Get** method of the **PersonController** Web API controller by using the relative URL **api/Person**. Running this code causes the text **Mike** to be displayed on the browser:

### Calling the Get Method

```
public async Task<IActionResult> Index()
{
 HttpClient httpClient = _httpClientFactory.CreateClient();
 httpClient.BaseAddress = new Uri("http://localhost:[port]");
 HttpResponseMessage response = await httpClient.GetAsync("api/Person");
 if (response.IsSuccessStatusCode)
 {
 Person person = await response.Content.ReadAsAsync<Person>();
 return Content(person.Name);
 }
 else
 {
 return Content("An error has occurred");
 }
}
```

## Passing Complex Objects to a Web API

In the previous example, you saw how a Web API can return a complex object to the client. It is also possible for a client to pass a complex object to a Web API. In order to do this, the object that needs to be sent must be serialized. The **HttpClient** class provides several methods to serialize the object, which include:

- **PostAsJsonAsync**. Sends a POST request to a specified URI with the given value serialized as JSON.
- **PostAsXmlAsync**. Sends a POST request to a specified URI with the given value serialized as XML.
- **PutAsJsonAsync**. Sends a PUT request to a specified URI with the given value serialized as JSON.
- **PutAsXmlAsync**. Sends a PUT request to a specified URI with the given value serialized as XML.

In the following example, you will first see a Web API controller that has a **Post** method that gets an instance of type **Entry** as a parameter. Then you will see how to call the **Post** method by using the **PostAsJsonAsync** method of the **HttpClient** class, which is used to serialize the parameter of type **Entry** to a JSON format.

The following code shows the **Entry** class:

### The Entry Class

```
public class Entry
{
 public string Key { get; set; }
 public string Value { get; set; }
}
```

The following code shows a Web API controller called **ValuesController** that has a **Post** method. The **Post** method gets an instance of type **Entry** as a parameter:

### A Web API that Gets a Complex Object

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 Dictionary<string, string> _items = new Dictionary<string, string>();

 public ValuesController()
 {
 _items["key1"] = "value1";
 _items["key2"] = "value2";
 }

 [HttpGet("{id}")]
 public ActionResult<string> Get(string id)
 {
 if (_items.ContainsKey(id) == false)
 return NotFound();

 return _items[id];
 }

 [HttpPost]
 public IActionResult Post(Entry entry)
 {
 if (_items.ContainsKey(entry.Key) == true)
 {
 return BadRequest();
 }

 _items.Add(entry.Key, entry.Value);
 return CreatedAtAction(nameof(Get), new { id = entry.Key }, entry);
 }
}
```

```
 }
}
```

The following code shows how to use the **HttpClient** class to call the **Post** method of the **ValuesController** Web API controller by using the relative URL **api/Values**. Running this code causes the text **succeeded** to be displayed on the browser:

### Calling the Post Method

```
public async Task<IActionResult> Index()
{
 HttpClient httpClient = _httpClientFactory.CreateClient();
 httpClient.BaseAddress = new Uri("http://localhost:[port]");
 Entry entry = new Entry() { Key = "key3", Value = "value3" };
 HttpResponseMessage response = await httpClient.PostAsJsonAsync("api/Values", entry);
 if (response.IsSuccessStatusCode)
 {
 return Content("succedd");
 }
 else
 {
 return Content("An error has occurred");
 }
}
```

## Demonstration: How to Call Web APIs by Using Server-Side Code

In this demonstration, you will first see how to register the **IHttpClientFactory** service. Then you will see how to inject the service to a controller. After that, you will see how to use the service to create an instance of type **HttpClient**. Finally, you will see how to use the **HttpClient** class to call the **Get** and **Post** messages of a Web API.

### Demonstration Steps

You will find the steps in the section “Demonstration: How to Call Web APIs by Using Server-Side Code” on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD13\\_DEMO.md#demonstration-how-to-call-web-apis-using-server-side-code](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD13_DEMO.md#demonstration-how-to-call-web-apis-using-server-side-code).

# Lab: Implementing Web APIs

## Scenario

You have been asked to create a web-based restaurant application for your organization's customers. To do this you need to create a page showing all the restaurant branches, enable users to order a reservation for a selected restaurant branch, add a wanted ad page, and allow submitting an application to a selected job.

You will create a server-side Web API application and a client-side ASP.NET Core MVC application. In the client-side application, you will call the Web API actions by using **HttpClient** and **jQuery**.

## Objectives

After completing this lab, you will be able to:

- Add actions to a Web API application.
- Call Web API actions by using HttpClient.
- Call Web API actions by using jQuery.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD13\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD13_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD13\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD13_LAK.md).

## Exercise 1: Adding Actions and Calling them by using Microsoft Edge

### Scenario

In this exercise, you will first add a controller and an action to a Web API application. You will then run the application and view the outcome by using Microsoft Edge. After that, you will add a controller and an action that gets a parameter. You will then run the application and view the outcome by using Microsoft Edge. Finally, you will add a Post action to the Web API application.

The main tasks for this exercise are as follows:

1. Add a controller and an action to a Web API application.
2. Run the application.
3. Add a controller and an action that gets a parameter.
4. Run the application.
5. Add a Post action to a Web API application.

## Exercise 2: Calling a Web API by Using Server-Side Code

### Scenario

In this exercise, you will call the Web API you developed in the previous exercise by using the **HttpClient** class. To do this, you will first register the **IHttpClientFactory** service in the **Startup.cs** file. You will then create an MVC controller and use the **HttpClient** class to call a **Get** action in the Web API. After that, you will create another MVC controller and use the **HttpClient** class to call a **Post** action in the Web API. Finally, you will add an action to the MVC controller in which you will use the **HttpClient** class to call a **Get** action in the Web API that gets a parameter.

The main tasks for this exercise are as follows:

1. Calling a Web API Get method.
2. Run the application.
3. Calling a Web API Post method.
4. Calling a Web API Get method that gets a parameter.
5. Run the application.

## Exercise 3: Calling a Web API by Using jQuery

### Scenario

In this exercise, you will call a Web API by using jQuery. You will first create an MVC controller and use jQuery to call a **Get** action in the Web API. After that, you will create another MVC controller and use jQuery to call a **Post** action in the Web API.

The main tasks for this exercise are as follows:

1. Calling a Web API Get method by using jQuery.
2. Run the application.
3. Calling a Web API Get method by using HttpClient.
4. Calling a Web API Post method by using jQuery.
5. Run the application.

**Question:** A member in your team noticed that when he runs the **Server** project of the application no browser is opened. Can you explain to him the reason for this?

**Question:** Your manager asked you to show the number of guests in the Reservation Information which is displayed when a user makes a new order. How can you achieve this?

## Module Review and Takeaways

You can use the Web API framework to facilitate creating REST-style calls in applications. REST-style 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. You can call a Web API by using server-side code and jQuery code.

### Review Question

**Question:** What are ASP.NET Core Web API controllers used for?

### Best Practice

Use **ActionResult<T>** or **IActionResult** to return a valid HTTP response message.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
A client that tries to call a Web API receives an exception with the message "Unable to connect to the remote server".	

# Module 14

## Hosting and Deployment

### Contents:

Module Overview	14-1
<b>Lesson 1:</b> On-Premises Hosting and Deployment	14-2
<b>Lesson 2:</b> Deployment to Microsoft Azure	14-19
<b>Lesson 3:</b> Microsoft Azure Fundamentals	14-29
<b>Lab:</b> Hosting and Deployment	14-42
Module Review and Takeaways	14-44

## Module Overview

Microsoft ASP.NET Core MVC applications are designed to simultaneously provide a service to multiple users with only a server installed, and the clients use their browsers to access the application. This results in applications that do not require the user to install dedicated software. This ensures that clients can access these applications on a wide range of devices.

To set up an ASP.NET Core application for a production environment, you will need to compile your code and compress it, set it up, and then run it on a dedicated server.

Hosting involves using a dedicated server to contain the compiled application and serve it to users as a web-based service. There are many different technologies that you can use to host your application, and you should choose one that is appropriate for your requirements.

Deployment is the process where the project code is compiled and then transferred to the hosting server by using a method that functions within the requirements of the hosting environment.

Microsoft Azure is a cloud service provided by Microsoft. It can be used to host ASP.NET Core applications and is a popular tool in the cloud technology market. It provides convenient web application related services with multiple billing options that can be selected according to user requirements.

### Objectives

After completing this module, you will be able to:

- Host and Deploy an ASP.NET Core MVC application on Internet Information Services (IIS).
- Host and Deploy an ASP.NET Core MVC application on Azure.
- Use Azure to improve the capabilities of your web applications.

## Lesson 1

# On-Premises Hosting and Deployment

While developing an ASP.NET Core application, a development environment based in Microsoft Visual Studio is perfectly reasonable for day-to-day development. However, for a testing environment that simulates production and for a production environment, you will need to be able to manage standalone copies of your application.

By deploying your application and creating a hosting environment you can start up a permanent server that can be accessed through a network, which allows multiple people to work on the application simultaneously. This is the final part of developing a web application because web applications, by definition, have to be accessible through a network.

In this lesson, you will learn about web servers and how to differentiate the web servers available in ASP.NET Core. You will then learn how to publish your existing ASP.NET MVC application by converting it to a collection of compiled and non-compiled files, which you can then use to publish to a web server. You will also learn how to set up hosting in an IIS environment, creating a permanent server to host your applications. Finally, you will also learn how to configure the application to safely interact with files used by your application.

It is important to understand that web applications rely on being accessible to users. This applies to applications such as in-company management applications that are designed to work in a closed environment and applications designed to work on the Internet. By deploying and hosting the application you can fulfill those requirements.

## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the server options provided by ASP.NET Core applications and their differences.
- Create and deploy a deployment build of an ASP.NET Core MVC applications.
- Host an ASP.NET Core application on IIS.
- Add important files, which can be accessed at run time, to your application.

## Web Servers

Being a web technology, ASP.NET Core application will require the setup of dedicated web servers to run. To set up web servers, you can choose from several different options based on your requirements.

ASP.NET core applications run on a process-based server implementation, which listens to requests from clients, and uses them to create an `HttpContext` object. This object is then injected into the middleware, and you can use it to control the flow of your application.

- Setting up the server is an important part of an ASP.NET Core MVC application and requires you to make important decisions
- By choosing Kestrel, you get:
  - A lightweight server that is fast
  - The ability to use reverse proxy
  - Cross-platform support
- By choosing HTTP.sys you get:
  - A robust framework with many prebuilt options
  - Windows-based authentication
  - Direct file transfer from the server

ASP.NET Core provides three server implementations:

- Kestrel (Cross Platform), which is the default server implementation.
- HTTP.sys (Windows) server implementation, which is based on the Kernel Driver and HTTP Server API
- Custom server implementation, which is based on Open Web Interface (OWIN).

By using one of the implementations, you can create and host a web server, allowing clients to consume your application.

Before jumping into Kestrel and HTTP.sys, we need to understand another important term in the world of web servers, reverse proxy server.

A reverse proxy server retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client as if they originated from the web server itself.

The benefits of reverse proxy include operating multiple different application servers on a single server, providing a firewall, limiting the exposure of application components, supporting load balancing architecture, and helping reduce distributed denial-of-service (DDOS) attacks.

## Kestrel

Kestrel is the default server implementation for ASP.NET Core, and it can be used with or without a reverse server proxy. Since Kestrel is designed to work cross-platform, it runs on its own process, allowing the use of technologies such as IIS, NGINX, or Apache. Those technologies cannot be used without Kestrel in ASP.NET Core.

Kestrel supports features such as HTTPS, WebSockets, and performance improvements while running on NGINX on Unix platforms. It is cross-platform and can be used across many different operating systems. It is a lightweight web server designed for efficiency and quick performance but does not support many features such as windows authentication, port sharing, direct file transmissions, and response caching.

To set up Kestrel, all you need to do is to call **WebHost.CreateDefaultBuilder(\*arguments\*)** in the **CreateWebHostBuilder** method of the **Program.cs** file. By default, all new ASP.NET Core applications are created with this configured, and the **CreateDefaultBuilder** method will call the **UseKestrel()** method behind the scenes.

The following example is a basic Kestrel web server configuration:

### Kestrel Web Server

```
public class Program
{
 public static void Main(string[] args)
 {
 CreateWebHostBuilder(args).Build().Run();
 }

 public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
 WebHost.CreateDefaultBuilder(args)
 .UseStartup<Startup>();
}
```

You can also further customize the Kestrel options by manually calling the **UseKestrel(\*Action<KestrelOptions>\*)** method as part of the **CreateDefaultBuilder** pipeline. You can configure many settings here. For example, by using the **Limits** property of the Kestrel Options object, you can set a variety of connection limits such as:

- **MaxConcurrentConnections.** Used to set how many connections the server can handle at a time.
- **MaxConcurrentUpgradedConnections.** Used to set the number of upgraded connections, such as those used in WebSockets.
- **MaxRequestBodySize.** Limits the size of the body in requests, set in number of bytes.
- **KeepAliveTimeout.** Sets the timeout for connections.

The following code demonstrates basic configurations for a Kestrel Server:

### Kestrel Server Configuration

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
 WebHost.CreateDefaultBuilder(args)
 .UseKestrel(options =>
 {
 options.Limits.MaxConcurrentConnections = 100;
 options.Limits.MaxConcurrentUpgradedConnections = 50;
 options.Limits.MaxRequestBodySize = 2 * 1024 * 1024;
 options.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(3);
 })
 .UseStartup<Startup>();
```

In this example, you can see that the Kestrel server is configured to support up to 100 concurrent connections, 50 upgraded connections (such as connections from web sockets), accepts requests of up to two MB, and times out connections after three minutes.



**Note:** There are many additional possible configurations for Kestrel according to requirements. To learn more, go to: <https://aka.ms/moc-20486d-m14-pg1>

## HTTP.sys

HTTP.sys is an alternative to Kestrel if ASP.NET Core is running on Windows. HTTP.sys supports Windows 7 or later and Windows Server 2008 R2 or later. It is a complete and robust web server implementation, which, unlike Kestrel, does not require external hosting solutions. However, it is not a lightweight solution.

HTTP.sys supports Windows Authentication, running multiple applications on the same port, HTTPS, Direct transfer of files, caching responses, and WebSockets. However, you can run **HTTP.sys** only in a Windows environment, and it does not support reverse proxy. It is also a heavy implementation, intended to handle all facets of a web server, and HTTP.sys does not work in conjunction with IIS or IIS Express.



**Best Practice:** For the purposes of compatibility and performance, we recommend using Kestrel as your web server. However, if a Windows Server is usable for your requirements and you require some of its features such as Windows Authentication, you can use HTTP.sys

In order to setup HTTP.sys, you will need to perform the following steps:

1. After calling the **CreateDefaultBuilder(\*arguments\*)** method in the application builder, you will need to pipe in a call to the **UseHttpSys()** as part of the builder. This will set up HTTP.sys with a basic configuration.
2. Add a new debug environment profile that does not use IIS or IIS Express, such as **project** in the project properties debug menu or in the **launchsettings.json** file. This was covered in Module 10, "Testing and Troubleshooting".
3. Select the new environment in the debug menu and start the application.
4. Go to "http://localhost:5000/". By default, the application will be hosted on port 5000.

After these steps are completed, your application will be running in HTTP.sys. However, you might want to configure additional parameters.

You can configure additional parameters by using an **Action** that returns an **HttpSysOptions** object. This object allows you to configure several options that can affect the behavior of the server. As an example, several properties that you can use include:

- **Authentication.** Exposes various sub-properties that can affect the application authentication, such as the **Schemas** property to set the application authentication schema.
- **MaxConnections.** Sets the maximum number of concurrent connections.
- **MaxRequestBodySize.** Sets the limit for the size of the body on requests.
- **UrlPrefixes.** Allows you to override the default URL.

The following code is an example of configuring HTTP.sys:

### Configuring HTTP.sys

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
WebHost.CreateDefaultBuilder(args)
 .UseHttpSys(options =>
{
 options.Authentication.Schemes =
Microsoft.AspNetCore.Server.HttpSys.AuthenticationSchemes.None;
 options.MaxConnections = 50;
 options.MaxRequestBodySize = 2 * 1024 * 1024;
 options.UrlPrefixes.Add("http://localhost:5250");
})
 .UseStartup<Startup>();
```

In this example, you can see that the application is set to not use an authentication schema, the application is also set to allow for up to 50 concurrent connections and accept requests with a body that is up to two MB in size. It is hosted with the URL `http://localhost:5250`. Attempting to access it with a different URL, including the default `http://localhost:5000` will fail.

 **Note:** There are many additional configurations, and some configurations can only be updated by using the Windows Registry. For more information go to: <https://aka.ms/moc-20486d-m14-pg2>

### Open Web Interface for .Net (OWIN)

OWIN is an open-source interface designed to operate between ASP.NET web servers and web applications. It was made with the intent to decouple the client and server architecture and allow for the development of simple modules.

 **Note:** OWIN is designed around ASP.NET rather than ASP.NET Core. Servers that run OWIN architecture will use ASP.NET. However, it is possible to create client applications, which interface with OWIN, in ASP.NET Core.

### ASP.NET Core Module

ASP.NET Core Module is an IIS module that enables you to set up a reverse proxy configuration for IIS applications. Being an IIS based module, it is incompatible with HTTP.sys, and you should use Kestrel instead. ASP.NET Core module runs by plugging into the IIS pipeline and redirecting requests to the appropriate backend application.

The ASP.NET Core module is responsible for managing the IIS process and runs in a separate process. When the initial connection occurs, the ASP.NET Core Module will start the kestrel application, and it will also restart it if the application has crashed.

Forwarding occurs in the ASP.NET Core Module when the client communicates with the IIS server that hosts the ASP.NET Core Module. The ASP.NET Core Module will then create an internal HTTP connection (it will use HTTP even when the connection with the client is HTTPS), to the ASP.NET Core application that runs on a Kestrel server. It will access the ASP.NET Core application by using an environment variable that is configured in the IIS startup. Kestrel will then handle the request in the normal flow before returning the response to the IIS server, which will forward it to the original client.



**Note:** The ASP.NET Core module is an IIS-based solution, which requires an IIS Server. If you work on a multiplatform or non-Windows environment, you can use Apache or NGINX instead. You can use either solution as reverse proxies when using a Kestrel server-based application.

## Running the server

After you set up the server, the final step is to actually run it.

In Visual Studio, you can do this by using the **Debug** menu. From this menu, select an environment profile and then start the application. This will run the application in either IIS Express, ASP.NET Core Module, or through the windows console. Running on separate environments was covered in Module 10, "Testing and Troubleshooting".

In Visual Studio Code, you can use the **Omnisharp** extension to run the application by using the CoreCLR debugger, which is an open source Microsoft .NET execution engine which is a part of .NET Core. This allows it to work cross-platform.

At times, you might not be able to run the application from an IDE. In such situations, you can use the various .NET CLI commands to run the application. You can run .NET CLI commands from various command line consoles such as Windows Command Prompt, Windows PowerShell, and Linux Terminal. You can usually add these commands to the system **PATH** by installing Visual Studio, but you can also add the commands by installing the .NET Core SDK, which is required for running .NET Applications. When not working on a development server, it is usually better to use the .NET Core SDK.

To run a server by using .NET CLI commands, you will need to browse to the project folder where the **.csproj** file is located. To run the commands, enter the **cd** command inside the command line console followed by the path to the folder that contains the **.csproj** file. The **cd** command exists in most common command line consoles.



**Note:** In Windows-based operating systems, the default drive is **C**. If your project is on a different drive, you can use the command, **\*drive letter\***: to switch drives. So, if your project is on drive **D**, you will use **d:** and so on.

To host a server, use the **dotnet run** command. This will run the build process, and then run the project in the current folder. This will allow you to run your project even without access to Visual Studio. This solution works across different operating systems. It can also be used alongside projects in Visual Studio Code, which allows you to bypass the need to install related extensions. Note that calling **dotnet run** will use any cached NuGet packages if they are available and will only download any which are missing.



**Best Practice:** While working on a production environment, it is best to use **dotnet run** because the faster setup is conducive to rapid testing. However, for production environments, it is best to use **dotnet publish**. This can allow you to publish once, create the necessary DLL files and ensure the latest relevant NuGet packages are used. Also, it is the recommended solution for hosting on production servers. You will learn about **dotnet publish** in the second topic of this lesson, "Hosting ASP.NET Core Application".

The following code is an example of dotnet run:

#### dotnet run

```
dotnet run
```

As an alternative option to navigating to the project file, you can choose to provide the **dotnet run** command a full file path to run from. This will remove the need to browse to the project folder.

## Hosting ASP.NET Core Application

One of the most important steps in the development of a web application is hosting it. Generally, web applications such as ASP.NET Core MVC applications need to be available to users and should be hosted on dedicated servers rather than on development environments.

In this topic, you will learn the basics of how to set up hosting for your ASP.NET Core MVC applications and learn about several of the many different technologies that you can use for your application.

The basic steps of hosting an application are as follows:

1. Publish the application. In this step, you will create a basic application. It will contain the DLLs and the resource files required to run your application, all inside a single folder, ready for deployment.
2. Set up a process manager. In this step, you will need to choose a dedicated process runner for your application, such as IIS or NGINX. This will keep the application up and handle application crashes and automatic startup.
3. Set up a reverse proxy configuration and load balancing. This is an optional step you can perform if you want the benefits of using a reverse proxy, which include protection from various security attacks (such as DDOS) and load balancing to support multiple servers.

For hosting ASP.NET Core MVC applications, you need to decide:

- What configurations are needed as part of the publishing process
- The server infrastructure to use to host your application
- Whether to use a reverse proxy as part of your application
- Whether load balancing functionality is required

### Publishing the Application

The first step in hosting an ASP.NET Core MVC Application is to publish the application. This will create a compiled copy of the application, alongside all required resource files such as JavaScript and HTML files, and store them in a single folder for ease of use. Publishing ensures that vital system files are not easily readable, while also being lighter than development files.

An important decision to make while publishing an application is to choose between a framework-dependent deployment (FDD), or a self-contained deployment (SCD).

If you use FDD, the deployment will not contain any .NET Core specific files because the files installed on the host will be used instead. You also do not have to specify a target operating system. This creates a smaller package, while also using the same .NET Core application installation as other applications on the server. However, using FDD requires the host to have the .NET Core version used by your application or a newer version, and if the .NET Core libraries are changed significantly enough in a future version, the application may start working in an unexpected manner.

If you use SCD, the deployment will contain a specific version of .NET Core, which is selected by you and will be packaged alongside your application. This ensures that you will not need to install it on the hosting server and your application will keep working even if the host updates its versions of .NET Core. However, because .NET Core is included as part of your package, you will need to select target platforms in advance, each additional platform increasing the installation size. This can particularly be troublesome on servers that run multiple .NET Core-based installations because a lot of space and memory in those servers might be taken up by multiple .NET Core installations that are running simultaneously.



**Note:** Note that any third-party files used by your application will also be present in both FDD and SCD deployments.

By default, a new project in a Visual Studio environment will be set with **Debug** configurations. However, builds that are intended for production environments will usually use **Release** configurations. You will need to set up the **Release** configurations to suit your need.

In order to set up the project for a publishing build in Visual Studio, in Solution Explorer, right-click the project, and then select the properties from the context menu. To update settings related to builds and publishing, click the **Build** tab. To publish for a production build, select **Release** from the **Configuration** drop-down list. This will ensure your project is set up for the release environment. Then go to the **Output** section of the **Build** properties. Inside the **Output path** box, set an output path of your choice or use the default one. This will determine where the compiled DLL files will be stored for the project. Note that it will add the version of the .NET Framework as a part of your path in the following format: "**\*your path\*\\*.NET Core version\*\**". Remember the final path because you will be using it later.

After this is done, browse to the project in your command line console of choice and call the **dotnet publish** command, just like you would call the **dotnet run command**. Rather than immediately running the application, it will get the latest version of all used NuGet packages and create a publish directory by using the directory you set in the **Output path** inside the **publish** folder. The resulting directory contains the compiled version of your application, and you can copy it to additional locations such as production servers.



**Note:** If you wish to deploy a self-contained installation, or use third-party libraries, additional steps will be required. For information, go to: <https://aka.ms/moc-20486d-m14-pg3>

To test your published build, you can call the **dotnet** command. The **dotnet** command accepts a path to a DLL file that contains a .NET Core application. It will then host the application in the command line console the command is called from. Remember that calling the **dotnet** CLI will require you to install the .NET Core SDK.

The following code is an example of dotnet publish:

### **dotnet publish**

```
dotnet publish
cd Publish\netcoreapp2.1\publish
dotnet MyApp.dll
```

In this example, you can see a project named **MyApp** being published. We publish it to the **publish** folder according to the build properties set for it. We publish it by calling the **dotnet publish** command under the **publish** folder. Then, we browse to the **publish** folder that has the compilation version for the project (in this case "**netcoreapp2.1**") and the output path which is set in the project properties, and then use the **dotnet** command with the DLL to run the application and ensure it is working.

 **Note:** The process described here covers an extremely basic publishing flow, and you may require additional settings and configurations. To read more about publishing using the .NET CLI go to: <https://aka.ms/moc-20486d-m14-pg3>

## **Setting up a Process Manager**

While you are technically able to keep the application running by using the **dotnet** command, the resulting server will be very basic and limited, and the application lifespan will need to be managed manually. This means that if the application crashes or the machine hosting it encounters a temporary issue, you will need to manually manage the process. This is obviously not a desirable circumstance.

To resolve these issues, you can use one of many dedicated process managers. These are programs such as IIS, NGINX, Apache, or Windows Services, all of which are designed to run applications and ensure they are kept running. Many of these are specifically designed to host web applications, but some such as Windows Services, are more generic options. In this module, you will learn about the basics of the most popular options.

### **IIS**

Internet Information Services (IIS) is a server architecture developed by Microsoft for handling requests. It is a popular option for maintaining web server processes, and natively supports many Windows-related technologies such as Windows Authentication. It is only supported in Windows Servers, and as such, you should validate your business requirements before using it to host your application.

IIS is highly flexible and manageable, allowing for features to be enabled and disabled as required. This can prevent issues caused because of applications running multiple unrequired features, leading to wasted resources.

 **Best Practice:** Being designed specifically for Windows Servers, IIS can be a good choice when the web servers are running Windows-based operating systems. However, it does not support Linux-based servers at all.

Deployment to an IIS Server will be covered in more depth in Topic 3, "Deploying to IIS".

### **NGINX**

NGINX is a process manager that is primarily used on Linux servers. It is a popular web server that can act as a standard server, as well as a reverse proxy server. In addition, it can act as a general usage TCP/UDP server. While you can run it on Windows servers, it is not optimized for it, and will generally underperform.

NGINX works by utilizing a single master process and multiple worker processes. The master process mostly handles the NGINX configurations and ensures that the configurations are used, as well as ensures that worker processes are maintained. In turn, the worker processes handle the actual processing of the various communication requests. The number of workers is dependent on the configuration file.

To find out more details on how to host ASP.NET Core applications on NGINX, go to: <https://aka.ms/moc-20486d-m14-pg4>

 **Note:** If you decide to use Linux-based hosting for your application, NGINX is a good choice to make because it is optimized to run on Linux. However, if a Windows Server is required, it is a poor choice and should be avoided.

## Apache

Apache is an open-source HTTP Server designed to operate effectively across multiple operating systems. It is designed to be flexible, highly compliant with technologies, and offers great extensibility through third-party modules because of it being an open-source tool. It is also constantly evolving, with new technologies being added on a frequent basis. Its main strength is the flexibility of use across different platforms.

The Apache HTTP server is configured by using various text files. These configurations further specify its behavior, enabling or disabling features as needed, including the behavior of third-party libraries that are in use.

For more details on how to host ASP.NET Core applications on NGINX, go to:

<https://aka.ms/moc-20486d-m14-pg5>

 **Note:** Apache can be a good server solution when multiple servers on different operating systems are required, such as when multiple clients require their own servers. You can learn how to deploy applications using Apache faster than learning how to deploy to both IIS and NGINX to deploy to different servers. In addition, Apache is the easiest solution when trying to support deployments across multiple different servers.

## Windows Services

A less common solution for a process manager is to host your application on a Windows Service. Unlike the previous options, Windows Service is not designed as a dedicated HTTP Server and therefore lacks many of the features common in other options. However, all installations of the Windows operating system include Windows Services as a core feature and as such, it can be used on any Windows installation.

When installing an ASP.NET Core application on a Windows Service, you can manage it through the Windows Services, enabling a relatively simple setup.

For more details on how to host ASP.NET Core applications on Windows Services, go to:

<https://aka.ms/moc-20486d-m14-pg6>

 **Note:** Windows Service hosting can be a reasonable choice if there is a requirement for minimum external requirements because it can work without additional applications. However, it is far less configurable than other options and requires Windows. Use this option when targeting Windows and IIS is not a viable option.

## Setting up a Reverse Proxy

An optional step in deployment is setting up your web server to use a reverse proxy. By using a reverse proxy, you can ensure a more secure and better load-balanced application at the cost of a more difficult setup process, higher latency due to an additional step in the navigation, and possible security issues with forwarding cookies between applications on the reverse proxy.

It is important to remember that using a reverse proxy will require you to use Kestrel and host using IIS, NGINX, or Apache. If you cannot fulfill these conditions, then you will be unable to gain the benefits of a reverse proxy.

 **Note:** The process for setting up the reverse proxy will vary between the different server types, and each one works differently.

When possible, it is almost always a good idea to use a reverse proxy, but you should analyze the specific requirements of your applications before making the decision.

## Load Balancing

Often, modern applications are designed to be scalable, leading to a requirement for load balancing. A load balancer is a system designed to direct incoming HTTP traffic between multiple servers. It will prefer sending new requests to servers with a lesser load at the time, allowing to optimize the user experience by providing a connection to available servers, while avoiding creating connections with busy ones.

If you use a reverse proxy, you might use one server to handle both load balancing and act as the reverse proxy. However, it is possible to keep these two systems separate, in which case, the load balancer will have to be between the application and the proxy server.

To learn how to add a reverse proxy server and implement load balancing, go to: <https://aka.ms/moc-20486d-m14-pg7>

## Deploying to IIS

One of the most convenient options for hosting an ASP.NET Core MVC application is to utilize IIS. You can easily set up IIS on Windows-based servers. As such IIS makes for a good host for your ASP.NET Core applications.

However, the process itself is not immediate, and you need to perform several important steps to correctly set up the server. In this topic, we will discuss the various steps and additional requirements to ensure that you can correctly deploy your application.

- Deploying an application requires a large investment at the start
- It becomes considerably easier on updates
- The first set up requires several steps:
  - Update the ASP.NET Core application to work with IIS
  - Set up IIS
  - Create the IIS Web Site
  - Deploy the application
- After the first deployment you can directly perform the deployment

 **Note:** Note that the instructions listed here facilitate a basic working IIS installation. You can set up many additional configurations based on what you need.

## Requirements in an ASP.NET Core Application

The first step in the set-up process occurs from within the ASP.NET Core application, and it is very likely that your application is already set up to handle the deployment. As part of the call to

**WebHost.CreateDefaultBuilder()** the **UseIISIntegration()** method is invoked, enabling Kestrel to integrate with IIS. If you prefer to build the **IWebHostBuilder** manually, you will need to call the **UseIISIntegration()** method to support IIS.

Another important step is to ensure that the **web.config** file is present in the published application. The **web.config** file is an important configuration file that is used by the IIS. These configurations determine the behaviors in working with the web application. Among other things, it tells the IIS to not serve some of the files present in the application root, causing potential security issues. It is important to ensure that the file is present in the published application root and to use the default name of **web.config**. If the file is renamed, the system will ignore it.

## Setting up IIS to Host ASP.NET Core Applications

You will need to ensure your designated host is able to run IIS to perform the deployment. In order to host, you will need to ensure IIS is installed on the designated computer.

 **Note:** Note that installing IIS may require administrative permissions and the process differs between Windows Server and Windows Home editions. If you encounter any permission trouble trying to set it up, you should ask your system administrators for help.

When IIS is set up, you will need to download and install the .NET Core Runtime. You can download .NET Core Runtime from the following link: "<https://www.microsoft.com/net/download>". You will need to install it after installing IIS. You will also need to reset the IIS process in order for it to load ASP.NET Core configurations. If you install .NET Core Runtime before you install IIS, you will need to rerun the .NET Core Runtime installer to repair the installation.

 **Note:** Ensure that you download .NET Core Runtime and not .NET Core SDK. It should appear next to the text "**Run Apps**", and the downloaded execution file should contain hosting as part of its name. Note that .NET Core SDK is not sufficient for IIS configuration.

## Creating the Web Site in IIS

Now that you have performed the initial preparations, you can start setting up the website in IIS. You have to perform the following steps:

1. Create a destination folder to host your ASP.NET Core MVC application. This folder will be used by IIS and will hold the physical files for your application.
2. Ensure that a logs folder is present. If it is not present, create one. This will ensure logs from the **stdout** will be added to the **Logs** folder. Alternatively, different configurations can be set from the **web.config** file.
3. Open the IIS Manager, and in the connection segment on the left, open the server node by clicking the arrow next to it.
4. Under the server node, right-click the **Sites** node, and then select **Add Website...**.
5. In the **Add Website** window, set the **Site name** to the name you wish to choose and set the **Physical path** to the folder you created in the first step. You can also set the bindings to use either HTTP or HTTPS as needed, set an IP address from the addresses available to the server and the port. By default, HTTP uses port 80, and HTTPS uses port 443. If you know what host name the application will

be served under, you can provide it now, if not you can change the host name later. After filling out your requirements, click **OK**.

6. At this point, if a default web site or another web site is already set up on your chosen port, you will get a notification message that only one site is supported for a single port. You can choose to either accept (you will only be able to run one site at once) or click **No** and change the port.
7. If you want the application to run in a process managed by ASP.NET Core instead of through IIS, you must also complete the following optional steps. This can help preserve the usage of specific versions of ASP.NET Core Runtime:
  - a. In the connection type, under the server node, select **Application Pools**.
  - b. Right-click the application you added and click the **Basic Settings...** option from the menu.
  - c. From the **.NET CLR version** drop-down list, select **No Managed Code**, and then click **OK**.

Once you have completed these steps, you have set up the IIS infrastructure for your application.

## Deploying the Application

The final step, which will be required whenever you update your application, is to deploy the application. A relatively simple way of doing this, which can be particularly useful in one-time deployments, is to copy the results of **dotnet publish** into the folder for the application. After this is done, you should stop and start the application in IIS, and then browse to it to verify the changes.

Alternatively, if you need to facilitate regular deployments, you might wish instead to use publish profiles in Visual Studio to automate the deployment process. These enable you to store various publishing properties, such as destination and automate several steps. To read more about publish profiles, go to: <https://aka.ms/moc-20486d-m14-pg8>

## File Providers

Sometimes your applications are reliant on physical files in the project to behave correctly. These can be additional configuration files, files used for storage, custom files for specific deployments, logging files for application log management and more.

In an ASP.NET Core MVC application, you can easily access any file or directory within the project structure by using file providers. This allows you to focus on the project structure itself, and not deal with the headaches of project structures, which may change as part of the build and publish process. In fact, both the **UseStaticFiles()** middleware and the Razor engine use the built-in File providers to locate resources, pages, and views.

In general, the primary interface you will use while working with files is **IFileProvider**. It provides methods for reading files and directories.

File Providers allow us to interact with actual files in the project structure:

- **PhysicalFileProvider** interacts with files that are physically present in the project structure
- **ManifestEmbeddedFileProvider** interacts with files which are embedded within the application itself, allowing for added security at the cost of being unchangeable at run time
- **CompositeFileProvider** allows us to combine two or more providers and use them all with a single interface

There are three separate implementations for **IFileProvider** available in ASP.NET Core applications:

- **PhysicalFileProvider.** Used to access files within the file system scoped to a specific root directory. This will be used for reading various non-compiled files such as XML, Text, and even HTML files.
- **ManifestEmbeddedFileProvider.** Used for accessing files embedded within the project assemblies. This can allow access to files that are normally embedded to the compiled assemblies of your ASP.NET Core application.
- **CompositeFileProvider.** Used for combining multiple other providers. This can allow you to use a single provider that can handle both compiled and non-compiled files.



**Note:** While using file providers, you must take great care with sending any information from files to the client. You have to make sure that no sensitive information is sent.

## PhysicalFileProvider

You can use the most commonly used file provider, **PhysicalFileProvider**, to read physical operating files such as XML configurations, log text files, JSON based resources, and media files.

**PhysicalFileProvider** uses the **System.IO.File** library for file management. However, it limits the file scope to a base directory and any children files and directories inside it. This makes it a safe option because it cannot access files from outside the project. Therefore, it cannot be used maliciously to cause harm to other applications. **PhysicalFileProvider** uses a directory path in its constructor to set the base directory, limiting access to subdirectories.

A convenient alternate option to instantiating a **PhysicalFileProvider** manually is to use the **IHostingEnvironment** service, which was covered in Module 10, "Testing and Troubleshooting". The **IHostingEnvironment** service exposes the **ContentRootFileProvider** property. This is a pre-instantiated **PhysicalFileProvider** originating in the application root. By using this, you can safely use a **PhysicalFileProvider**, which is already configured to use the root of the application.

The following code is an example of getting **PhysicalFileProvider** from **IHostingEnvironment**:

### PhysicalFileProvider from IHostingEnvironment

```
private readonly IFileProvider _fileProvider;
public HomeController(IHostingEnvironment hostingEnvironment)
{
 _fileProvider = hostingEnvironment.ContentRootFileProvider;
```

After obtaining a **PhysicalFileProvider**, you can then use the **GetFileInfo(\*filePath\*)** method to get an **IFileInfo** object, which is used for interacting with files. The file path parameter supplied to **GetFileInfo** should direct to the file you wish to read.

In order to read the actual file, you will need to use the **CreateReadStream()** function on the **IFileInfo** object. This will create a memory stream designed to hold the content of the file as it is read. It is advised to encapsulate this step in a using statement, as the stream implements **IDisposable** and creates a connection to the file.

After the stream is established, you will need to use it to instantiate a new **StreamReader(\*stream\*)** object, which is used for performing the actual reading. It requires the previously created stream as a parameter.

Finally, you can use the various functionalities of the **StreamReader** to read the file. For example, **ReadToEnd()** will read all segments of the stream, which have not yet been read.

The following code is an example of using a file provider to read a text file:

### Reading a Text File with **IFileProvider**

```
public IActionResult Index()
{
 string content;
 try
 {
 IFileInfo fileInfo = _fileProvider.GetFileInfo("appText.txt");

 if(fileInfo.Exists)
 {
 using (Stream fileStream = fileInfo.CreateReadStream())
 {
 StreamReader streamReader = new StreamReader(fileStream);
 content = streamReader.ReadToEnd();
 }
 }
 }
 catch (Exception)
 {
 content = string.Empty;
 }
 return Content(content);
}
```

In this example, you can see that a file provider is used to read the "**appText.txt**" file from the root directory and return its content.



**Best Practice:** Due to a dependency on sources which are external operations to the application, the methods for interacting with files should be encapsulated with a try/catch block. This can help prevent unexpected failures.

### ManifestEmbeddedFileProvider

Another useful option for a file provider, particularly when security is a big concern, is using **ManifestEmbeddedFileProvider**. Compared to **PhysicalFileProvider**, this file provider offers an option to use files specifically embedded into the application DLL. This can act as an additional layer of security, allowing you to add sensitive files inside the application and use them internally or expose specific segments as needed.

The first part of this process is to set up the project configuration to support manifest embedded files. By right-clicking the project and selecting **Edit \*Project file\***, you can edit the project configuration. Under the **PropertyGroup** element, you will need to add a new **GenerateEmbeddedFilesManifest** element. Inside the element, set the value to **true** to support embedding files into the manifest.

You will then need to find the **ItemGroup** element. Inside it, you can add **EmbeddedResource** tags to specify files that will be embedded inside the application. Each **EmbeddedResource** must contain the **Include** attribute, and the value provided for the attribute needs to be a valid file path. You can either use the wildcard **\*** in your paths, or you can provide specific file paths, so you could use the string **".txt"** to embed all text files in the root directory, or you can provide a specific file.

The following code is an example of a **csproj** file with embedded resources:

## Embedding Resource Files

```
<Project Sdk="Microsoft.NET.Sdk.Web">

 <PropertyGroup>
 <TargetFramework>netcoreapp2.1</TargetFramework>
 <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
 </PropertyGroup>

 <ItemGroup>
 <Folder Include="wwwroot\" />
 </ItemGroup>

 <ItemGroup>
 <EmbeddedResource Include="*.txt" />
 <PackageReference Include="Microsoft.AspNetCore.App" />
 <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.0" />
 </ItemGroup>

</Project>
```

Once this is done, you can create a new **ManifestEmbeddedFileProvider(\*assembly\*)** to receive a file provider. To get the root of the current assembly, you can use **System.Reflection.Assembly.GetEntryAssembly()** as the **assembly** parameter, which will be used by the **ManifestEmbeddedFileProvider** to act as a file provider for embedded files. This will allow you runtime access to files that are not actually present as physical files.

The following code is an example of **ManifestEmbeddedFilerProvider**:

### ManifestEmbeddedFileProvider

```
private readonly IFileProvider _fileProvider;

public HomeController()
{
 _fileProvider = new ManifestEmbeddedFileProvider(Assembly.GetEntryAssembly());
}

public IActionResult Index()
{
 string content;
 try
 {
 IFileInfo fileInfo = _fileProvider.GetFileInfo("embeddedText.txt");
 if(fileInfo.Exists)
 {
 using (Stream fileStream = fileInfo.CreateReadStream())
 {
 StreamReader streamReader = new StreamReader(fileStream);
 content = streamReader.ReadToEnd();
 }
 }
 }
 catch (Exception)
 {
 content = string.Empty;
 }
 return Content(content);
}
```

Note that only the method for instantiating the file provider differs from using physical file providers.

 **Note:** It is important to note the distinction that embedded files are added as part of the assembly at the compilation stage, while physical files remain separate from the assembly.

## CompositeFileProvider

Occasionally, you may end up in situations where you need both external resources (such as configuration files) and embedded files (files with sensitive data). In such a case, rather than separately maintaining multiple separate file providers inside your application, you can instead use the **CompositeFileProvider** to unify several different providers.

 **Note:** Note that you are not limited to using **CompositeFileProvider** to merging **PhysicalFileProvider** and **ManifestEmbeddedFileProvided**. You can also use it to merge physical file providers that rely on separate file paths.

In order to set up **CompositeFileProvider**, you will need to first set up at least one of the providers it will use. As such, it is recommended to set up **CompositeFileProvider** as a service in the **ConfigureServices** method. To do this, create a new **CompositeFileProvider** and provide it with an array of file providers. Note that because **IHostingEnvironment** is not injected into the **ConfigureServices** method, you will need to add a **constructor** to the **Startup** class and add it as a property. Once the composite provider has been instantiated, use **services.AddSingleton<IFileProvider>(compositeProvider)** to add it as a service.

The following code is an example of setting up **CompositeFileProvider** as a service:

### Add CompositeFileProvider as a Service

```
private readonly IHostingEnvironment _hostingEnvironment;

public Startup(IHostingEnvironment hostingEnvironment)
{
 _hostingEnvironment = hostingEnvironment;
}

public void ConfigureServices(IServiceCollection services)
{
 IFileProvider physicalProvider = _hostingEnvironment.ContentRootFileProvider;
 IFileProvider manifestEmbeddedProvider =
 new ManifestEmbeddedFileProvider(Assembly.GetEntryAssembly());

 IFileProvider compositeProvider =
 new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

 services.AddSingleton<IFileProvider>(compositeProvider);
 services.AddMvc();
}
```

 **Best Practice:** Note that you can add any of the three provider types as an injectable service in a similar way. However, only a single provider can be used for injection because they all use the same interface. It is therefore always preferable to inject the **CompositeFileProvider** whenever multiple providers are used in the application.

For the next step, inject the composite file provider into the desired location, and then use it to interact with files in the same way as the other providers.

The following code is an example of using a **CompositeFileProvider**:

### Using CompositeFileProvider

```
private readonly IFileProvider _fileProvider;

public HomeController(IFileProvider compositeProvider)
{
 _fileProvider = compositeProvider;
}

public IActionResult Index()
{
 string content;
 try
 {
 IFileInfo embeddedFileInfo = _fileProvider.GetFileInfo("embeddedText.txt");
 if(embeddedFileInfo.Exists)
 {
 using (Stream fileStream = embeddedFileInfo.CreateReadStream())
 {
 StreamReader streamReader = new StreamReader(fileStream);
 content = streamReader.ReadToEnd();
 }
 }

 IFileInfo physicalFileInfo = _fileProvider.GetFileInfo("physicalText.txt");
 if(physicalFileInfo.Exists)
 {
 using (Stream fileStream = physicalFileInfo.CreateReadStream())
 {
 StreamReader streamReader = new StreamReader(fileStream);
 content += streamReader.ReadToEnd();
 }
 }
 }
 catch (Exception)
 {
 content = string.Empty;
 }
 return Content(content);
}
```

Note that you can see the difference by changing the files during runtime and refreshing. If a physical file is changed or removed, the application will throw an error or update after a refresh. Meanwhile, if an embedded file is removed from the project structure, it will still keep working.

## Lesson 2

# Deployment to Microsoft Azure

Microsoft Azure is a cloud service provided by Microsoft for organizations to build and manage solutions easily. This service has multiple hardware and software abstraction levels, which include:

- **Infrastructure** as a service (IaaS). Virtualization of physical infrastructure components, which means virtual machines being provided as a service.
- **Platform** as a service (PaaS). Platform-level resources that provide a running environment to run a deployed application, such as application servers, which are provided as a service residing on the cloud.
- **Software** as a service (SaaS). Using existing software functionality through services residing on the cloud.

In this lesson, you will learn about various options available on Azure to deploy your web applications. The options available cover the entire spectrum from a complete-do-it-yourself system (**Virtual Machines**) to a completely managed platform as a service (**App Service**). You will learn the advantages and limitations of the various options. You will learn how to deploy your application by using **App Service**, which is a fully managed platform for application deployment that provides automatic scalability and infrastructure maintenance.

Azure also provides tools for developers to be able to remotely debug applications deployed on the cloud. **Azure Application Insights** provides developers with the ability to perform in-depth monitoring of their application. Starting from request rates and response times to performance counters, such as CPU, memory, and network usage, developers can monitor various metrics. Application Insights provides live streaming of exception traces, a snapshot of data assembled from live operations, and log traces that allow developers to diagnose and resolve issues quickly.

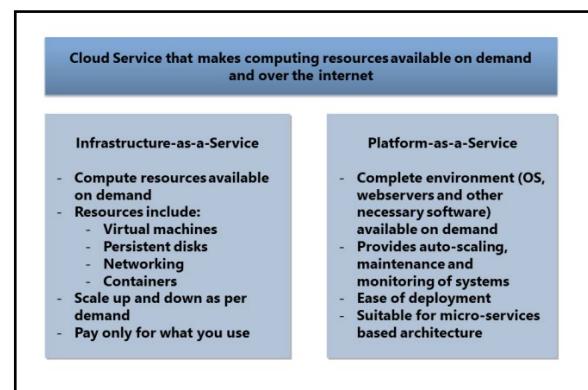
### Lesson Objectives

After completing this lesson, you will be able to :

- Describe Microsoft Azure and its benefits.
- Deploy a completed MVC web application to Microsoft Azure.
- Use Application Insights to resolve issues.
- Understand various deployment options, deployment slots and the benefits of using them.

### What is Microsoft Azure?

Microsoft Azure is a cloud service provided by Microsoft. A “cloud” makes computing resources available on-demand and over the internet. The computing resources include servers, storage, networking, and databases. Azure also makes lots of software and platforms available as resources. These resources are available on-demand and billing is as per use.



The most basic of these services is Infrastructure as a Service (IaaS). This allows you to use any kind of IT infrastructure, such as servers, persistent disks, networks, and operating systems without the hassle of having to buy any hardware or configure any machines. You can provision these machines over the internet at the click of a button. Once you are done using them, you can “delete” them, again at the click of a button. You will be charged for these machines only for the duration that you used them. Some of the services available under IaaS in Azure are:

- Virtual Machines. Provision Windows and Linux virtual machines in seconds
- Container Instances. Easily run containers with a single command
- Traffic Manager. Route incoming traffic for high performance and availability
- Azure DNS. Host your DNS domain in the Cloud

On the other hand, Platform-as-a-service (PaaS) is an advanced service that provides a complete environment to run your applications. In the case of IaaS; deployment, maintenance, monitoring, and scaling of resources to run your application is your responsibility. In the case of PaaS, the only thing you need to do is deploy your application. The platform provides servers with pre-installed software such as any middleware and development tools needed to run your application. Features such as scalability, high-availability, and multi-tenant capability are provided by the platform, reducing the amount of coding that developers must do. The need for additional resources to monitor, update, and manage the applications is also reduced because the platform provides these as built-in services. PaaS services provided by Azure include:

- App Service. Quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform
- Service Fabric. Develop microservices and orchestrate containers
- Azure SQL Database. Managed relational SQL Database as a service
- Azure Functions. Process events with serverless code
- Azure Storage. A massively scalable object store for data objects, a file system service for the cloud, a messaging store for reliable messaging, and a NoSQL store.

These are just a few examples from a wide range of services that Microsoft Azure provides for applications to be built, deployed, and run in a highly-available, scalable, and resilient manner.

## Benefits of Hosting in Microsoft Azure

Deploying applications on Azure has a number of benefits. While ease of use and cost-effectiveness are the most obvious benefits, there are a number of other advantages that make hosting in Azure particularly useful.

### Flexibility and Elasticity:

Provisioning and using resources is extremely simple with Azure. In Azure, the resources needed, such as virtual machines, can be provisioned in seconds and it offers a range of options to provide a better fit for your needs.

Azure offers users the ability to scale from one to thousands of virtual machines. With data centers across the world, you can scale to where your customers are.

#### Benefits of Azure:

- Efficiency
  - With PaaS, deploying and scaling application is very easy. This leads to efficient use of resources.
- Elasticity
  - Ability to scale up to thousands of machines
- Security
  - Common compliances and certifications in place. Also provides DDoS protection, threat protection, and information protection.
- Cost
  - Pay as you use model means no up-front investment needed and hence more cost-effective.
- Developer Tools
  - A wide variety of tools available for building and deploying your application automatically.

## Security and Compliance:

Security is integrated into every aspect of Azure. Data encryption at rest is available for services across the SaaS, PaaS, and IaaS cloud models. Azure supports various encryption models, including server-side encryption that uses service-managed keys, customer-managed keys in Key Vault, or customer-managed keys on customer-controlled hardware. With client-side encryption, you can manage and store keys on-premises or in another secure location.

Services such as Azure DDoS protection and Azure Advanced Threat Protection help you keep your applications and systems secure from any malicious or suspicious user or device activity.

Azure has compliances for most of the regulations across the world. The following are a few of them:

- Federal Financial Institutions Examination Council (FFIEC)
- Health Insurance Portability and Accountability Act (HIPAA)/Health Information Trust Alliance (HITRUST)
- Payment Card Industry Data Security Standard (PCI DSS)
- Federal Risk and Authorization Management Program (FedRAMP)

## Developer Tools and DevOps:

Azure has a wide range of tools that integrate completely with Azure to make building, managing, and continuously delivering cloud application very simple. Azure supports all the commonly used third-party DevOps tools such as Ansible, Chef, Terraform, Puppet for automated infrastructure management.

Azure DevOps Services allows developers to create a complete continuous integration/continuous delivery (CI/CD) pipeline by allowing them to automate building, testing, and deploying applications. Azure provides complete integration with all commonly used source repositories such as Git and Bitbucket. Azure application Insights provides developers with the ability to get real-time metrics from production systems, helping them resolve issues quickly. To summarize, all the tools needed for all life cycle stages of an application are fully integrated with Azure. This makes application management easy and fast.

## Cost Effectiveness:

The most obvious cost-benefit of Azure is that there no upfront payment required for any of the infrastructure or services being used. The pay-as-you-use model ensures that you pay for the resources only for the duration that you use them. However, there are additional pricing models, where you can opt for committed usage and get up to 70% discounts. Azure also offers special Dev/Test pricing for running your development and QA workloads.

## Deploying Web Application on Microsoft Azure

To deploy a web application on Azure, there are various options available:

- Azure Virtual Machines. This is the most basic service available. With this, the entire responsibility of installing and configuring all required software, monitoring application, scaling application as needed must be done manually. This option is best for legacy applications.
- Azure App Service. Azure App Service is the best choice for most web apps. Deployment and management are integrated into the

### Azure App Service

- PaaS from Microsoft Azure, provides a managed platform to deploy and host your applications
- Offers auto-scaling, high availability, and load balancing
- Provides App Insights to monitor application performance
- Applications are hosted on Microsoft's global infrastructure
- App Service is ISO, SOC, and PCI compliant
- App Service provides easy integration to other Azure services such as Storage, Active Directory etc.
- Integrates with various tools for continuous deployment

platform, sites can scale quickly to handle high traffic loads, and the built-in load balancing and traffic manager provide high availability.

- Service Fabric. Service Fabric is a good choice if you're creating a new app or re-writing an existing app to use a microservice architecture. Apps, which run on a shared pool of machines, can start small and grow to massive scale with hundreds or thousands of machines as needed.

In this section, we will work with Azure App Service and deploy web applications to Azure App Service.

The simplest and easiest way to deploy ASP.NET Core applications to Azure is by using **Visual Studio Publishing Wizard**.

To deploy your application, in Solution Explorer, right-click the project, and then select **Publish**. In case you are deploying your application for the first time, you will be presented with the **Pick a publish target** dialog. Use the default value, **App Service**. Make sure **Create New** is selected and then click **Publish**. A new dialog will appear, which displays the details to create a new app service on Azure. This is a one-time activity and in subsequent deploys, you just select the service you have already created.

In the **Create App Service** window, if you have not already done so, sign in to your Azure account by using the **Add an account** button in the top-right corner. You will be prompted for an **App Name**. App names are globally unique and an immediate check to verify the uniqueness of your app name is performed. If the name is already used, a notification will appear and you will have to choose a new name.

Choose your subscription, this subscription determines the access to resources in Azure. You will also be required to choose a **Resource Group**. A resource group is used to group related resources so that they can be managed as a group. This is useful to automate monitoring, deployment, etc. If you already have a resource group, choose that. You can also create a new resource group by, choosing the **New** option.

Next, you need to choose the **App Service Plan**. An App Service plan defines a set of compute resources for a web app to run. The plan specifies the region where you want to host your application. Ideally, this should be where most of your customers are located. The plan also helps you choose the size of the virtual machine instance, number of virtual machines, and pricing tier. The pricing tier governs how scaling is handled. If you do not have a plan, you can create a new one, and then select that.

Once all the necessary options are chosen, a summary of what's going to happen appears at the bottom of the window. Make sure everything is as you planned.

Click **Create**. This will create all the necessary resources in Azure. Right after this step, Visual Studio builds your project and deploys it. Once deployment is completed, a new tab opens in your browser, and will automatically point you to your deployed application. The URL to access your application is in the form: "http://<APP-NAME>.azurewebsites.com" where **APP-NAME** is the name you entered while publishing your application.

You now have a working ASP.NET Core Application inside Azure.

## Demonstration: How to Deploy a Web Application to Microsoft Azure

In this demonstration, you will learn how to deploy your ASP.NET web application to Azure App Service from Visual Studio, also how to view the details of the app service in Azure Portal

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Deploy a Web Application to Microsoft Azure" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD14\\_DEMO.md#demonstration-how-to-deploy-a-web-application-to-microsoft-azure](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD14_DEMO.md#demonstration-how-to-deploy-a-web-application-to-microsoft-azure)

## Azure Deployment Strategy

**The deployment process** is the process by which the code for your application is built and deployed to the servers that will host it and eventually be available for all the users.

**The deployment strategy** is the strategy that you choose for the deployment to happen. There are various ways to deploy to Azure and it all depends on your specific scenario:

### Visual Studio Publishing Wizard

The easiest way to deploy to Azure is by using Visual Studio Publishing Wizard, as demonstrated in the previous lesson. Even if you've only started with Azure, with few simple clicks your Web App is up and running.

### FTP

In order to use the FTP option, you need an existing Web App inside Azure. To use it you need to know your hostname and credentials, which can be found inside Azure, where your Web App resides. After you have the credentials, you can use your favorite FTP client to deploy the site directly to Azure.

### Azure CLI

The Azure CLI tool lets you make deployments straight from your console. In concept, this is similar to what the publishing wizard does, except this happens through a command.

While all the above options are fast and easy, they are obviously not suitable to deploy production-level applications. Some of the drawbacks of this kind of deployment are:

- Such deployments generally require a downtime wherein the old application version is stopped and the new version is installed and started.
- Such deployments are high risk since if there are any issues, the impact is felt directly on the live system.
- In case there are any problems with the deployment, there is no easy way to roll back. The rollback is also a manual process mostly and adds to the downtime.

To handle these issues, Azure provides different deployment strategies.

### Staging Environment, Deployment Slots, and Swapping

In general, a good practice is to deploy your changes or application to an environment that is as close to the production environment as possible. Such environments are called staging environments. Once an application is deployed to a staging environment, tests can be run on this system and after all tests pass, the application can then be deployed to production.

Azure provides a feature called deployment slots which makes this possible very easily and efficiently. Once you deploy your application to App Service, you can create a deployment slot. This deployment slot is a replica of your original deployment environment. The original deployment environment is also called the production deployment slot.

At the time of creating a deployment slot, you need to specify a name for the slot. This gets appended to the **App Name** and is used to create the URL to access your particular deployment.

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Traditional Deployment Options           <ul style="list-style-type: none"> <li>▪ FTP, CLI, Visual Studio – simple, but not suitable for production grade applications</li> <li>▪ Require downtime and it is not easy to rollback in case of problems</li> </ul> </li> <li>• Deployment Slots           <ul style="list-style-type: none"> <li>▪ Provides additional environments similar to production environments</li> <li>▪ Applications can be deployed to these for testing or staging</li> <li>▪ Allows traffic routing for a percentage of the incoming traffic to one of the deployment slots to perform A/B testing</li> <li>▪ On successful testing, slots can be swapped</li> </ul> </li> <li>• Resource Templates           <ul style="list-style-type: none"> <li>▪ Ability to define resources needed for an application as a template</li> <li>▪ Re-create the entire stack from this template</li> </ul> </li> </ul> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

For example, if your app name is "**MyAzureApp**", the URL to access the original deployment will be "http://MyAzureApp.azurewebsites.net". Now if you create a deployment slot with the name "**beta**", the URL to access the deployment in this slot will be "http://MyAzureApp-beta.azurewebsites.net". So, in short, you now have two different deployment environments that are identical to each other but have their own hosts and configurations.

Once you have additional slots, you can use these to run all kinds of testing on the staging environment. After all tests have successfully passed, you can swap your environments. Once you swap, your application that was deployed to the staging deployment slot will now be available in the production deployment slot and vice-versa. Internally this is achieved by swapping the virtual IP addresses of the source and destination slots, thereby swapping the URLs of the slots.

All of this is achieved without any downtime. It is also possible to attach certain settings to slots. Database is a good example; your production and staging environments will connect to different databases. So, database connection settings can be attached to their own slots. So, while your application gets swapped, they continue to connect to the correct database.

If after swapping development slots, you think that there are any issues with production deployment, you can always roll back your deployment by doing another swap. This reverses the original swap. All of this makes transitioning from staging to production environments very smooth and easy, enabling your deployments to remain robust.

### Azure Resource Manager Templates

A production application almost always has more than the Web App itself. It may have a database, a caching layer, and other resources. When we want to deploy the site to production, we need to take care of these resources too. Moreover, what happens if everything gets deleted by accident? What if we want to deploy the same stack in a new region?

We can solve all these issues by using the Azure Resource Manager and Azure Resource Manager Templates.

This template is basically a JSON document in which you declare imperatively all the resources you need, and the dependencies between them. Using the template, Azure Resource Manager can deploy, update, or delete all the resources for your solution in a single, coordinated operation. You use a template for deployment and that template can work for different environments such as testing, staging, and production. Resource Manager provides security, auditing, and tagging features to help you manage your resources after deployment.

You can also use deployment slots in conjunction with Azure Resource Manager by specifying the type of deployment as slots and giving the name of the slot to which the deployment is done.

## Debugging a Microsoft Azure Application

For any application running in production mode, the ability to constantly monitor the application, get insights into the performance of the application, and be able to quickly diagnose any problems is a very important requirement. Azure Application Insights provides tools for all these requirements.

### Application Insights Configuration

Let us look at how you can use this feature for analysis and monitoring of your application deployed on Azure.

- Application Insights – ability to monitor applications running in App Service.
- Application Insights provides
  - Metrics – such as CPU usage, memory consumption, page views, performance of events etc
  - Live Stream – ability to see key metrics in a streaming fashion.
  - Analysis from Visual Studio
- Remote Debugging – ability to debug live applications from Visual Studio
- Server Explorer – Ability to manage Azure Services from Visual Studio

To start using this service, you will need to do the following:

1. Create an Application Insights Resource in the Azure Portal. In the portal, click **All Resources**, select **Management tools**, and in the list that appears on the right, select **Application Insights**.
2. In Visual Studio, go to Solution Explorer, select the project folder, right-click and choose **Add**. In the options, choose **Application Insights Telemetry**.
  - a. In the **Application Insights Telemetry** configuration window for, register your application for Application Insights. Provide your Azure account details and choose the Application Insights resource created in Step 1.
  - b. At the bottom of this window, you will see the **Add SDK** option. This will add the Application Insights SDK to your solution. This will help you see live streaming data on the portal.

With this you have now configured your application to be used with Application Insights. This allows you to see various analytics of your application in the Azure Portal. In addition, you can also explore telemetry data from your application running in Azure App Service directly in your Visual Studio.

We will now look at how to use Application Insights.

### Analysis from the Azure Portal

In the Azure Portal, from **All Resources**, select the **Application Insights** resource created earlier. The **Overview** page shows you a few default graphs such as

- Failed Requests. This shows the number of requests made to the application that returned an error response.
- Server Response Time. The time it took for the server to respond to a request.
- Server Request Count. The number of requests made to the server
- Page Load Time. The time it takes for a page to load.

In the **Investigate** section on the left-hand side, you will see several features that will help in application monitoring.

**Application Map** helps you spot performance bottlenecks or failure hotspots across all components of your distributed application. This view displays all the components that are part of the application. In the case of complex applications, this is very helpful in visualizing the system and quickly spotting the problem areas. Clicking on a component gives more details about that component and you can Investigate Further for more analysis of that component.

**Live Metrics Stream** shows you key metrics of your application in a streaming fashion. This is helpful for monitoring your application during specified periods to understand live application usage patterns and behavior.

**Metrics Explorer** allows you to create your own graphs. Application Insights telemetry collects various metrics from your application. Metrics Explorer allows you to create charts for any metric you wish to track. You can create charts with specific granularity, specific metrics, and aggregations. You can also set **alerts** from the Metrics Explorer. To create an alert, you need to do the following:

1. Choose the metric
2. Choose the condition
3. Choose the threshold
4. Choose the period for which the condition violation should occur before raising an alert.
5. Choose the action to be taken. The action can be an email or an Http/Https request to a webhook.

The metrics explorer has a wide range of performance counters available for you to monitor. These are categorized into:

- **Usage**. This gives you information about the usage of your application in terms of events, page views, sessions, users, etc.
- **Failure**. This gives you information about different failures, how many were server exceptions, browser exceptions etc.
- **Performance Counter**. Such as available memory, CPU counters, request rate, etc.

**Availability** allows you to create availability tests. In these tests, Application Insights sends requests to endpoints configured by you. If the application does not respond or responds slowly, an alert is triggered. There could be multiple tests that can be configured for an application, from a simple index page URL to dependent requests. A Dependent Requests test parses embedded links and images and performs requests for each of these as well. It is also possible to set up a multi-step test, where a series of URLs representing various stages of a workflow can be configured.

**Failures** section shows the details of failed requests. This view can be used to get more details about the failures such as response codes, the exception being thrown, etc.

**Performance** section gives you details of the time taken for various requests to get completed. This gives you information about the most occurring requests, the average time taken by them, etc.

## Telemetry Analysis from Visual Studio

As mentioned earlier, once you have configured your application to provide Application Insights Telemetry, in addition to the Azure Portal, you can also view the telemetry data in Visual Studio.

For this analysis, go to **Application Insights Trends**. To reach this window, on the **View** menu, click **Other Windows**. If you are coming to this window for the first time, you will be asked to **Select the Application Insights Resource**. Subsequently, Visual Studio remembers the last resource used and displays data for that resource.

The **Application Insights Trends** window lets you analyze telemetry based on:

- **Type**: This is the type of data you want to analyze, such as server requests, exceptions, dependencies, page views, etc.
- **Time Range**: You can further split your data to a specific time range.
- **Groups**: The data you want to analyze can be grouped on different parameters such as performance, response code, failure types etc. This grouping is dependent on the type of data you are analyzing.

For example, if you are analyzing page views, you can group the data by Performance, Page names, Application Version, etc.

- **View:** This lets you decide whether you want to view data points as counts or percentages as well as whether anomalies (data that is out of normal range) should be shown.

Once all the selections are done, click **Analyze Telemetry**. This will result in a time series histogram being shown in the top half of the screen followed by tables with dimension counts in the bottom half. You can use the dimension counts for further filtering of data.

If you have chosen to view the data with anomalies, the histogram will show the anomalies in red. The size of the circle shows the size of the anomaly. You select the anomaly to get details such as the deviation.

Double-clicking this will take you to **Application Insights Search** view. This lets you get into details of the instance, the request and other fine-grained information of the anomaly.

In this view, you can choose custom time filters and a combination of data types. Since the time range is custom, this gives you the ability to view time series over a longer period and then narrow down to a smaller range, by clicking the histogram bars. The bottom part of the window lets you further filter data with a large number of options. You can also see details for each data point (each request, each page view) to further drill down and get to any problem areas. For example, if you are looking at page views, the bottom view will give you a list of page views. Clicking on any of these views gives you further details of that particular view or request. These details include browser version, the location of the request, the instance details that handled the request and many more.

On top of this view, you will see the **Track Operation** tab. Clicking this will show you the breakup of time taken in various subrequests. This is a very useful view to drill down into to find the time taken by various components and comes in handy for performance tuning.

## Remote Debugging

You can also debug your application running on Azure App Service. To be able to do this, you need to do two things:

1. Deploy the app to Azure App Service with **Debug** configuration.
2. Attach a debugger to the deployed app

Let us discuss these steps in detail, starting with attaching a debugger to the deployed app. We can use two Visual Studio views to perform this task: Cloud Explorer and Server Explorer.

## Cloud Explorer

The **Cloud Explorer** view provides an overview of your Azure resources and resource group, allowing you to perform resource related diagnostics actions. In Visual Studio, from the **View** menu, go to **Cloud Explorer**. In this window, you should see your Azure subscription. In case you do not, click **Account Management** and login to Azure from Visual Studio. Once you are logged in, you should see all your Azure services in **Cloud Explorer**. Select the **App Services** option and choose your application deployment. In the bottom half of the window, you should see the **Actions** panel. In this panel, click **Attach Debugger**. Once this is done, you can create breakpoints in your code and debug just like your local application. You can step through functions, examine data, etc.

 **Note:** Use remote debugging for applications in production with a lot of caution. Step-through and breakpoints could impact the overall performance of applications in production. Also, debugging in this manner means that all events and data are streamed to your local Visual Studio. This increases network usage and can impact billing.

## Server Explorer

The **Server Explorer** view allows you to manage your web app running in Azure App Service from Visual Studio. You can check your applications configuration, settings, etc. from this view. You can navigate to **Server Explorer** from the **View** menu. If you are not already connected to Azure, connect through the **Server Explorer** window. Once connected, this window shows you the relevant Azure services. Under App Service, you will see the relevant resource groups. Expanding a resource group will show the application under this resource group. Select your application, right-click, and choose **View Settings**. This will show you the configuration of your application for Azure App Service such as whether Remote Debugging is turned on, the connection strings configured for this application, etc.

The server explorer also allows you to attach a debugger. From **Server Explorer**, you can also view all the files uploaded to App Service. Under your Web App, if you expand **Files**, you can see all the files uploaded to the app service including the config files. You can edit these files to change log levels, debug settings, etc. and the changes will take effect immediately. This is a very powerful option to be able to see the exact configuration that your application is running under

## Lesson 3

# Microsoft Azure Fundamentals

In the previous chapter, we have had a brief introduction to Azure and seen how to deploy your applications to Azure and monitor them. In this chapter, we look at some more services of Azure that can be used to build scalable, distributed, and robust applications.

We first look at the various options available to store and access data from your application. Depending on your application needs, you can choose the correct option to store data. Since Microsoft Cloud Storage is a managed service, using this eliminates the need to worry about local storage, backup etc.

We also look at some key points that need to be considered while building distributed applications and the options that Azure provides to handle these, such as caching as a service, mechanism for deferred processing, etc.

In some situations, due to regulatory or other concerns, part of applications may have to run on local datacenters. Azure provides Azure Stack that enables services, similar to Azure to run on a local data center.

## Lesson Objectives

After completing this lesson, students will understand:

- Details about related services in Azure.
- The choice of storage options and which options suit which conditions.
- What is a distributed application and the considerations while building a distributed application.
- How to use Azure Stack to build hybrid applications.
- How to use Key Vault to store secrets and how to access them through an application.

## Microsoft Azure Storage

Azure Storage is a managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. All access to data in storage happens through storage accounts. An Azure storage account provides a unique namespace to store and access your Azure Storage data objects. There are various types of storage available.

### Azure File Share

Azure Files offers fully managed file shares in the cloud that are accessible via the industry standard Server Message Block (SMB) protocol.

Azure file shares can be mounted concurrently by cloud or on-premises deployments. This is a good option in systems that have traditional file servers and need to be migrated to the cloud.

- Managed service providing storage that is highly available, secure, durable, scalable, and redundant.
- Type of storage :
  - **Azure Blob storage** - object-based storage; can be used to store image files, audio and video clips.
  - **Azure File Share** - fully managed file shares in the cloud
  - **Azure Queue storage** - service for storing large numbers of messages
  - **Azure Table storage** - service that stores structured NoSQL data in the cloud

## Azure Queues

Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls by using HTTP or HTTPS. The most common usage of queues is for asynchronous communication between applications. A queue can contain a set of messages. Each message can be up to 64KB and can remain in the queue for no more than 7 days.

## Azure Table Storage

Azure Table storage is a service that stores structured NoSQL data in the cloud, providing a key/attribute store with a schema-less design. Since Table storage is schema-less, it's easy to adapt your data as the needs of your application evolve. Access to Table storage data is fast and cost-effective for many types of applications and is typically lower in cost than traditional SQL for similar volumes of data.

## Azure Blob Storage

Blob storage is object-based storage. This can be used to store image files, audio and video clips, log files etc. Blob storage is optimized for storing large amounts of data – both text as well as binary. Some of the most common uses of Blob storage are for backup and restore and archiving, storing files for distributed access. Data stored in Blob storage can also be used for analysis by any analytics systems. It is possible to configure objects to be stored in Azure storage to be directly accessed by using HTTP or HTTPS. This can be used to load static images from websites. Loading images this way improves the performance of websites.

To start using Blob storage, first you need to create a storage account. Each storage account has its own endpoint. There are different types of storage accounts such as General Purpose storage accounts and blob storage accounts. Blob storage account is a specialized account in which you can specify the access tier. The access tier is chosen based on the frequency of blob access.

Within a storage account, you can create containers. A container organizes a set of blobs, similar to a folder in a file system. There can be multiple containers within a storage account and each container can store multiple blobs. Each blob stored in a container in a storage account has its own unique URL. This URL is of the form:

`http://<storageaccountname>.blob.core.windows.net/<containername>/<blobname>`

All the above storage options have their own APIs.

To use these storage options in your .NET applications, you need to add the storage service as a Connected Service. To do this, in Solution Explorer, right-click your project, and click **Add**. Choose **Connected Service**. In the box that appears, choose **Cloud Storage with Azure Storage**. This will then prompt you to create a storage account if you don't already one or choose from the ones you have. Once done, the required Nuget packages will get installed and you can now work with any of the above storage options by using the appropriate library.

To access blobs from an application, you first need to access the storage account. To access the storage account, you need a connection string, this can be obtained from the Azure portal. From the storage account, a client needs to be created. From the client, we need to get a reference to the container and from the reference to the container, you can upload and manage blobs. It is also possible to stream content into blobs by using the API.

The following code is an example of using Azure Blob storage:

### Azure Blob Storage

```
const string connectionString = "<REPLACE WITH YOUR CONNECTION STRING>";

//Get a reference to Storage Account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

//Get a reference to a container and create the container if it does not exist.
CloudBlobContainer container = blobClient.GetContainerReference("mydemocontainer");
await container.CreateIfNotExistsAsync();

//Get a reference to the blob. The blob gets created if it is not present.
CloudBlockBlob blob = container.GetBlockBlobReference("hello.txt");

await blob.UploadTextAsync("Hello World");
```

## Demonstration: How to Upload an Image to Microsoft Azure Blob Storage

In this demonstration, you will learn how to create a storage account and a container by using Azure Portal. Also, you will upload an image to the container and then you will create a container from the web application. Finally, you will upload an image to the container from the application.

### Demonstration Steps

You will find the steps in the section "Demonstration: How to Upload an Image to Microsoft Azure Blob Storage" on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD14\\_DEMO.md#demonstration-how-to-upload-an-image-to-microsoft-azure-blob-storage](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD14_DEMO.md#demonstration-how-to-upload-an-image-to-microsoft-azure-blob-storage)

## Microsoft Azure SQL

Azure SQL is a managed relational database as a service provided by Azure. When applications are running on Azure, it follows that the database should also be running on Azure. Azure SQL provides an easy and efficient way to use a database running on Azure. With Azure SQL, there is no need to manage the underlying infrastructure of the database. All patching of the code, maintaining the infrastructure and other management is handled by Microsoft and this happens seamlessly.

Azure SQL Database – a fully managed SQL database

- Infrastructure management taken care of by Azure
- Scalability – Allows for dynamic scalability, i.e. increase/decrease infrastructure configuration without a downtime
- Availability – provides automatic backups, replication and failure detection
- Security and Compliance – provides data encryption at rest and in transit. Provides access control and tools to protect sensitive data.
- Intelligent insights and monitoring – provides automatic performance monitoring and tuning.

### Scalability Options

One of the critical requirements of applications running in production mode is to be able to scale quickly. Azure SQL provides a few options based on the deployment model. Azure SQL provides two deployment models:

- Database Transaction Unit (DTU) based. This deployment model has a set of pre-defined compute sizes – that include storage, IOPS, CPU and backup size. There are different tiers of DTU that give the users choices depending on workloads to be run.
- vCore based. In this deployment model, the user can choose CPU, memory, and storage based on requirements.

In both the deployment models, Azure SQL provides the ability to dynamically scale – the ability to go to a higher tier in case of DTU or higher configuration in case of vCore deployment model, without any downtime.

## Availability Options

To be able to run your application 24/7, the underlying infrastructure and services should run at the highest availability. Azure provides a whole lot of features to support this. This includes automatic backups, replications, failure detection handling underlying hardware, software and network failures. Azure SQL also provides active geo-replication allowing you to configure up to four read replicas of the database in any of the Azure data centers.

## Intelligent Insights and monitoring

Azure SQL provides automatic performance monitoring and tuning. Azure SQL learns about your database and the queries being executed and based on this provides performance tuning recommendations. It can also automatically monitor SQL database performance at scale and notifies users of performance degradation issues, identifies the root cause, and suggests improvements and fixes for them. With automatic tuning, users can also ask Azure SQL to directly apply tuning recommendations. Azure SQL, with this feature, ensures your database is carefully monitored and compared before and after every tuning action, and if the performance doesn't improve, the tuning action is reverted.

## Security and Compliance

Azure SQL provides all the basic security measures and compliances. By default, data encryption is provided for data in transit as well as at rest. However, users can also choose to use their own keys for encryption and choose different encryption keys for specific columns. One of the most important security measures is to protect sensitive data such as personal identification, financial data, etc. Azure SQL provides advanced capabilities for discovering, classifying, labeling, and protecting the sensitive data in databases. SQL Database secures your data by limiting access to your database using firewall rules, authentication mechanisms requiring users to prove their identity, and authorization to data through role-based memberships and permissions, as well as through row-level security. Azure SQL also provides a centralized security dashboard by using the SQL vulnerability assessment tool that can help discover and track potential database vulnerabilities.

## Working with Azure SQL

To create an Azure SQL database, you need to login to Azure portal and go to **Sql Databases** and create a server. This is where the database will be hosted. You can choose the configuration to suit your needs. At the time of server creation, you also need to specify the username and password for the server admin login. After the server is created, you can create a database.

Accessing the database is like accessing a SQL database installed on any server and all the usual tools will work. In addition, it is also possible to execute queries from the Azure portal.

The only change needed in your applications to work with Azure SQL is to use the connection string for your SQL instance in Azure. You can obtain this connection string from the Azure portal. In the portal, navigate to the **SQL databases** page and click the database you want to work with. In the **database** page, in the left-hand menu, click **Connection Strings** which will show you the connection strings to be used for various connection options in the right-hand side. Copy the one that is relevant to you and use it in application settings.

In most situations, you would want to use a local database during testing and Azure SQL Database for production. You can configure this at the time of publishing. At the time of publishing the application to Azure, in the **Publish** window, click **configure**. In the **Configure** page, click **Settings** and then expand the **Databases** section. Select **Use this connection string at runtime** check box and type the connection string for Azure SQL here.

## Design a Distributed Application by Using Microsoft Azure

With cloud; designing, deploying and maintaining a distributed application is fairly simple. A distributed application consists of an application running on one or more servers. A distributed application can scale much better since there are multiple servers available to handle requests. A distributed application also makes your application highly available. The multiple servers in a distributed environment are behind a load balancer. The load balancer directs requests to one of the many servers, either round-robin fashion or based on the load on a server.

- Distributed applications – ability to auto scale.

- Need for centralized session management
  - Session-affinity
  - Redis based session management.
- Need for asynchronous communication between components for easy scalability
  - Azure Service Bus – integrated message broker
  - WebJobs – ability to perform background tasks
  - Azure Functions – run small functions on the cloud.
- Hybrid applications – run on Azure and on-premise datacenters
  - Azure Stack – run Azure Services on datacenters in any location

When you are deploying your application to Azure App Service, you have the option of turning on **autoscaling** for your application. Once autoscaling is enabled, depending on the server load, Azure App Service will create more instances running with your application. The load balancer will automatically be configured to start sending requests to the new servers.

While autoscaling is very efficient and reduces the overhead of manually monitoring the server utilization and taking action once capacity is reached, it is important to consider the impact of a distributed environment on the application itself. For example, most applications save some form of user data in the user session. The common practice is to store user session In-proc – i.e. in the memory of the server where the application is running. However, the drawback of this is that if the server crashes, all the user session data is lost. In a distributed environment an additional consideration is determining which server has the user session data for a particular session. This is important because if the user session data is stored on one server, all requests related to that user must be sent to that server only. This is also called **session-affinity**. This is achieved by use of cookies and storing the server information in a cookie. Load balancers will examine this and redirect the request to the server with the session data.

Obviously, session-affinity has an impact on scalability. If most requests are tied to a particular server, even though additional servers are added, they will not be utilized properly. One solution to this is to store session data in such a manner that all servers can access it. There are two options for this. The session data can be stored in a database or a common cache from where it can be accessed by all servers. While storing data in a database is the most secure of all options, in terms of performance it is not the best. That leaves us with the option of using a common cache.

Azure Cache for Redis provides a session state provider that you can use to store your session state in-memory with Redis Cache instead of a SQL Server database. To use the caching session state provider, first, configure your cache, and then configure your ASP.NET application for the cache by using the Redis Cache Session State NuGet package.

You can create a Redis cache from the portal and provide a DNS name and resource group name. The DNS name is used to access the cache. Configuring Azure Cache for Redis can also be done from the Azure portal. Some properties that can be configured are the cluster-size, eviction policy to free up cache space, firewall, access control etc.

On the application side, use the Nuget Package Manager to install `Microsoft.Web.RedisSessionStateProvider`. The NuGet package downloads and adds the required assembly references and adds the necessary section into your **web.config** file. This section contains the required configuration for your ASP.NET application to use the Redis Cache Session State Provider. Also, remove the standard InProc session state provider section in your **web.config**.

Once this is done, your application is ready to use Redis Cache for user session data storage.

Caching strategies will also be discussed in detail in a separate topic.

## Deferred Processing

To build highly scalable distributed architecture, one key consideration is to have loosely coupled components and to make use of deferred processing where ever possible. Such systems need ways in which components can communicate without being dependent. Azure provides various services for this.

One such service is Azure Storage Queues which has been discussed in the section on storage earlier. A more advanced option that is available is Azure Service Bus.

**Microsoft Azure Service Bus** is a fully managed enterprise integration message broker. Service Bus is most commonly used to decouple applications and services from each other and is a reliable and secure platform for asynchronous data and state transfer. Data is transferred between different applications and services by using *messages*. A message is in binary format, which can contain JSON, XML, or just text. Azure Service Bus supports queues as well as topics. While a queue is often used for point-to-point communication, topics are useful in publish/subscribe scenarios. Listed below are some important features of Azure Service Bus:

- Scheduled Delivery. Ability to submit messages for delayed processing. For example, at midnight every day.
- Batching. All messages sent by a client in a certain period of time are grouped in a single message.
- Transactions. Supports grouping operations against a single messaging entity. For example, several messages sent to one queue from within a transaction scope will only be committed to the queues log when the transaction completes.
- Auto-forwarding. Enables chaining a queue or subscription to another queue or topic that is part of the same namespace.

**Microsoft Azure WebJobs** is a feature that allows for the creation of programs/scripts that can run as background tasks within your application context. WebJobs is a feature of an Azure App Service and runs in the same instance as your application. WebJobs can be run in the following ways:

- Continuously. Starts as soon as the application starts.
- Triggered. Runs when this is triggered explicitly.
- Scheduled. Runs as per a schedule given at creation.

**Microsoft Azure Functions** is a solution for easily running small pieces of code or functions, in the cloud. Unlike an application, this is not meant to be an always running website, but a function that gets executed at the occurrence of any particular event. The events or triggers can be:

- `HttpTrigger`. Trigger the execution of your code by using an HTTP request.
- `TimerTrigger`. Execute cleanup or other batch tasks on a predefined schedule.
- `BlobTrigger`. Process Azure Storage blobs when they are added to containers.
- `QueueTrigger`. Respond to messages as they arrive in an Azure Storage queue.

The above are a few examples of triggers and there are more such triggers. With these, your application can trigger a function and it can get executed asynchronously.

## Hybrid Environments

In some situations, for regulatory reasons or data location concerns, it might not be possible to use public cloud such as Azure to host your applications and store data. For such situations, Azure provides Azure Stack. Azure Stack is a hybrid cloud platform that lets you provide Azure services from your datacenter. Azure Stack is an extension of Azure. With Azure Stack, developers can now build modern applications across hybrid cloud environments, balancing the right amount of flexibility and control. Developers can build applications using a consistent set of Azure services and DevOps processes and tools, then collaborate with operations to deploy to the location that best meets the business, technical, and regulatory requirements. In short, use of Azure Services from an application development point of view will remain the same, while deployment of the application can happen either on Azure, or on-prem with Azure Stack. Some common use cases where hybrid deployments using Azure Stack are:

- Cloud applications that meet varied regulations: Customers can develop and deploy applications in Azure, with full flexibility to deploy on-premises on Azure Stack to meet regulatory or policy requirements, with no code changes needed. Financial and banking related applications are the ones where regulatory requirements are very strict and can benefit from Azure Stack.
- Edge and disconnected solutions: Customers can address latency and connectivity requirements by processing data locally in Azure Stack and then aggregating in Azure for further analytics, with common application logic across both.
- Cloud application model on-premises: Customers can use Azure services, containers, serverless, and microservice architectures to update and extend existing applications.

Azure Stack has two deployment options:

**Azure Stack Integrated Systems.** Azure Stack integrated systems are offered through a partnership of Microsoft and hardware partners, creating a solution that offers cloud-paced innovation and computing management simplicity. Because Azure Stack is offered as an integrated hardware and software system, you have the flexibility and control you need, along with the ability to innovate from the cloud. Use Azure Stack integrated systems to create new scenarios and deploy new solutions for your production workloads.

**Azure Stack Development Kit (ASDK).** ASDK is a free single server deployment that's designed for trial and proof of concept purposes. The portal, Azure services, DevOps tools, and Marketplace content are the same across ASDK and integrated systems, so applications built against the ASDK will work when deployed to an integrated system.

Some of the features of Azure Stack are:

- Azure Stack makes all the same services available in local data centers that are available in Azure. These include IaaS (Virtual Machines, Storage, Networking etc.), PaaS (Azure App Service, Azure Functions etc.).
- The user experience for Azure Stack and Azure remains the same, with the same portal being available in both scenarios. The same tools – for monitoring, deployment, configuration and administration work on Azure as well as Azure Stack. This makes working in hybrid environments very easy and hassle-free.
- Any innovations in Azure, including new Azure services, updates to existing services, and additional Azure Marketplace applications are continuously delivered to Azure stack. This helps in keeping all environments in sync all the time.
- You can use the same identities (for users as well as applications) across Azure and Azure Stack, since both work with Active Directory.

So, to conclude use of Azure Stack enables you to use the same scalable, flexible, and modern cloud services that Azure provides with the choice of using them in a data center and location of your choosing, such that any regulatory, latency needs are met.

## Design a Caching Strategy

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. In situations where data is mostly static and there is a latency in accessing data from the original data store, using a caching mechanism will show dramatic performance improvements. Caching data from a persistent store, such as a database also helps since most databases have a limit on the number of concurrent connections that are available.

- Azure Cache for Redis
  - Redis cache as a service
  - Globally available, hence suitable for distributed applications
  - Useful for caching data within an application – session data, data that needs to be fetched from a database, etc.
- Azure CDN – Content Delivery Network
  - Global caching for static content such as html files, images etc.
  - Files are cached in edge servers across the globe
  - Files are served from closest location to consumers, thus improving performance

It is important to decide what kind of data is to be cached and how the data in the cache should be handled. The most obvious type of data is static data such as configuration information, reference information, etc. While caching is ineffective for highly dynamic data, since the data can get stale very quickly, it is possible to cache only a portion of the data that is not updated frequently. In the case of highly dynamic data, caching is still possible if the data is non-critical.

Another important aspect of caching is how the expiration of data is handled and how data is kept up-to-date. There are various expiration policies such as:

- A most-recently-used policy (in the expectation that the data will not be required again).
- A first-in-first-out policy (oldest data is evicted first).
- An explicit removal policy based on a triggered event (such as the data being modified).

The main options available on how to cache data is an in-memory store, local file system storage, or a shared global cache.

**An in-memory cache** is a primitive option and is good only for stand-alone applications. In a distributed application, each instance would end up having its own copy of cache data leading to synchronization issues. Also, the size of such a cache is limited to available memory. This limitation can be overcome by using a local file system, however, this introduces a read-write overhead which makes this type of cache slower.

**Using a shared global cache** can help alleviate concerns that data might differ in each cache, which can occur with in-memory caching. Shared caching ensures that different application instances see the same view of cached data. An important benefit of the shared caching approach is the scalability it provides. Many shared cache services are implemented by using a cluster of servers and utilize software that distributes the data across the cluster in a transparent manner. An application instance simply sends a request to the cache service. The underlying infrastructure is responsible for determining the location of the cached data in the cluster.

**Azure Cache for Redis** is an implementation of the open source Redis cache that runs as a service in an Azure datacenter. It provides a caching service that can be accessed from any Azure application. Azure Cache for Redis is a high-performance caching solution that provides availability, scalability, and security. It typically runs as a service spread across one or more dedicated machines. It attempts to store as much information as it can in memory to ensure fast access. It is compatible with many of the various APIs that are used by client applications.

You can provision a cache for your application by using the Azure portal. When you create a Redis cache from the portal you need to provide a DNS name and resource group name. The DNS name is used to access the cache. Using the Azure portal, you can also configure the eviction policy of the cache, and control access to the cache, monitoring and alerting policies, firewall rules, etc.

In an earlier section, we have already seen how to use Redis as a session state provider. In addition to this, Azure Cache for Redis can be used generally for normal caching needs as well. Redis is a key-value store, where values can contain simple types or complex data structures such as hashes, lists, and sets. Keys can be permanent or tagged with a limited time-to-live, at which point the key and its corresponding value are automatically removed from the cache.

To add Redis to your application, in Visual Studio, use NuGet Package Manager to install StackExchange.Redis.

The following code sample is a snippet that shows how to connect to Azure Cache for Redis and access data from the cache:

### A controller class

```
ConfigurationOptions config = new ConfigurationOptions();

config.EndPoints.Add("<CACHENAME>.redis.cache.windows.net");
config.Password = "<Redis cache key from management portal>";

//Replace CACHENAME with the DNS Name you provided while creating the cache.

var cm = ConnectionMultiplexer.Connect(config);
var db = connection.GetDatabase();
// You now have a connection to the cache. You can now use this connection, to add data
to the cache or // to retrieve it. Some examples are:
db.StringSet("key", "value");
var key = db.StringGet("key");
```

For more information on API, refer to <https://aka.ms/moc-20486d-m14-pg9>

### Content Delivery Network

While Redis cache is suitable for caching data used within an application, an application front end has a lot of static files such as HTML files, images, logos, etc. that can also benefit from caching. This type of caching is possible with Content Delivery Network (CDN). A CDN is a distributed network of servers that can efficiently deliver web content to users. Azure CDN is a global CDN solution for delivering high-bandwidth content. With Azure CDN, you can cache static objects loaded from Azure Blob storage, a web application, or any publicly accessible web server, by using edge servers in point-of-presence (POP) locations that are close to end users, to minimize latency.

The benefits of using Azure CDN to deliver web site assets include:

- Better performance and improved user experience for end users, especially when using applications in which multiple round-trips are required to load content.
- Large scaling to better handle instantaneous high loads, such as the start of a product launch event.
- Distribution of user requests and serving of content directly from edge servers so that less traffic is sent to the origin server.

To add Azure CDN to your web application, in the Azure portal, navigate to your app in the **App services** page. In the **App services** page, in the **Settings** section, go to **Networking**. On the right-hand side pane, expand the **Azure CDN** section and in this, you will see the **Configure Azure CDN for your app** option. In the **Azure Content Delivery Network** page, provide the **New endpoint** settings. You will need to specify:

- Name of your CDN
- Pricing Tier - Specifies the provider and available features.
- CDN endpoint name - Any name that is unique in the azureedge.net domain. Cached resources are accessed as <endpointname>.azureedge.net

After entering the above information, click **Create** and the CDN profile gets created.

After this, you can access all your static content by using <endpointname>.azureedge.net and this will be served from the CDN.



**Note:** It takes a while for the CDN registration to take effect, so you might not see performance improvement immediately.

That CDNs are best for static files is now established, Azure CDN provides an option to improve the performance of web pages with dynamic content. This can be achieved with Azure CDN dynamic site acceleration (DSA) optimization. This option can be chosen while creating the CDN endpoint.

One important consideration while using CDNs, is to be aware of the effects of cached files. In case your application makes changes to any static files, they may not get reflected. Azure CDN edge nodes will cache assets until the asset's time-to-live (TTL) expires. After the asset's TTL expires, when a client requests the asset from the edge node, the edge node will retrieve a new updated copy of the asset to serve the client request and store the new copy inside the cache.

The best practice to make sure your users always obtain the latest copy of your assets is to version your assets for each update and publish them as new URLs. CDN will immediately retrieve the new assets for the next client requests. Sometimes you may wish to purge cached content from all edge nodes and force them all to retrieve newly updated assets. This might be due to updates to your web application or to quickly update assets that contain incorrect information.

## Security in Microsoft Azure

While deploying applications on the cloud and running them in a distributed environment, one key thing to consider is securing your application configuration or application secrets. i.e. database connection strings, third-party API keys, passwords, etc. It is possible to do so by storing these in configuration files in an encrypted manner. Managing keys is an overhead. Managing change in these – for example, change of a password/connection string– could possibly mean redeployment of the application and is not ideal.

- Azure Key Vault – a secure store for keys and secrets
- All information within a vault is encrypted and stored.
- Useful to securely store application configuration information such as connection strings, passwords etc.
- Application access to Key Vault is through authentication by Azure Active Directory.
- Key Vault is backed by HSM.

Azure provides Azure Key Vault that can be used by applications to store keys and secrets. Key Vault is a cloud service that works as a secure secrets store. Key Vault helps safeguard cryptographic keys and secrets used by cloud applications and services. By using Key Vault, you can encrypt keys and secrets (such as authentication keys, storage account keys, data encryption keys, and passwords) by using keys protected by hardware security modules (HSMs).

Using Key Vault provides the following benefits:

- Centralize application secrets: Centralizing storage of application secrets in Key Vault allows you to control their distribution. This greatly reduces the chances that secrets may be accidentally leaked. When using Key Vault, application developers no longer need to store security information in their application. This eliminates the need to make this information part of the code.
- Securely store keys: Secrets and keys are safeguarded by Azure by using industry-standard algorithms, key lengths, and HSMs.
- Monitor access and use: Since keys and secrets are stored centrally, it is easy to monitor how and when these are being used.
- Simplified administration: The overhead of having to encrypt data, perform key management by rotating keys periodically, having to manage an in-house HSM, certificate management, etc. is removed by Key Vault.

To use Key Vault, the following steps need to be performed:

1. Create a Key Vault.
2. Authorize your application to access Key Vault
3. Generate a token to be used by your application for authenticating itself with the Key Vault
4. Create a Key Vault client to access keys and secrets.

We will look at each of these steps in details.

## **Creating a Key Vault**

To create a key vault, go to the Azure portal and in the search bar on top, search for "**key vault**", and then go to the **Key Vault** page. On this page, click **Create**. On this page, provide the requested details. You will need to specify a globally unique name, choose an appropriate Azure subscription, resource group, and location for the vault. After entering all relevant information, click **Create**. Your vault has now been created. The URL to access the vault will be **<https://<YOUR VAULT NAME>.vault.azure.net>**. The vault is now empty.

Keys and secrets can be added to the vault by using the portal, CLI, or API.

## **Authorize your application to access Key Vault**

Before you can start accessing the key vault from your application, you will need to authorize the application to access the vault. For this, you need to do two things:

- Create an identity for the application in Azure Active Directory (AD).
- Provide this identity in Key Vault to authorize the application to access Key Vault.

Azure Active Directory (AD) is an identity service provider responsible for verifying the identity of users and applications that exist in an organization's directory and ultimately for issuing security tokens upon successful authentication of those users and applications. Key Vault uses this service to authorize applications.

To create an identity in Azure AD, in the portal search for "**App Registrations**" and on this page click **New application registration**. On the **Create** page provide your application name, and then select **WEB APPLICATION AND/OR WEB API** (the default) and specify the URL to your web application. Click the **Create** button. When the app registration is complete, you can see the list of registered apps which will also contain your application. The list will display the **Application ID**. If you click your application, a pane opens on the right-hand side and will have a **Settings** button on top. Click this button and in the pane below, click **Keys**. Enter in a description in the **Key description** box and select a duration, and then click **SAVE**. The page refreshes and now shows a **key value**. The application Id and key are the identity of your application.

The next step is to configure Key Vault to allow your application access. For this, in the portal, go to your key vault. From the menu, choose **Access Policies** and click **Add New**. In the window that appears, click **Select Principal**. In the list that appears, select the name with which you had created an identity for your application in AD. Also select the appropriate key, secret and certificate permissions necessary for your application. Generally, **GET** operations will be the most often performed operations. Click **Save** and with this, your application is now authorized to access Key Vault.

### **Generate token to be used by your application for authenticating itself with the Key Vault**

Once the above configuration is done, you are now ready to start accessing the keys and secrets from the Key Vault. We now look at the changes needed in your application to perform these operations. There are two packages that your web application needs to have installed.

- **Active Directory Authentication Library** has methods for interacting with Azure Active Directory and managing user identity.
- **Azure Key Vault Library** has methods for interacting with Azure Key Vault.

These can be installed using the NuGet package manager.

To use the Key Vault API, you need an access token. The Key Vault client handles calls to the Key Vault API but you need to supply it with a function that gets the access token.

The following code sample shows the function needed to get the access token:

#### **Util.cs**

```
//the method that will be provided to the KeyVaultClient
public static async Task<string> GetToken(string authority, string resource, string scope)
{
 var authContext = new AuthenticationContext(authority);
 ClientCredential clientCred = new ClientCredential(WebConfigurationManager.AppSettings["ClientId"],
 WebConfigurationManager.AppSettings["ClientKey"]);
 AuthenticationResult result = await authContext.AcquireTokenAsync(resource,
clientCred);

 if (result == null)
 throw new InvalidOperationException("Failed to obtain the JWT token");

 return result.AccessToken;
}
```

In the above snippet, the **ClientId** and the **ClientKey** are as obtained from the Azure AD. A general best practice is to add this configuration in **web.config** and further to provide actual values for these from the Azure portal. This can be done in the **Application Settings** for your application in **App Service**.

## Create a Key Vault client to access keys and secrets.

The next thing to do is to start accessing data from the Key Vault. For this, you need to create an instance of **KeyVaultClient**. A reference to the **GetToken** method needs to be passed while doing this.

The following code sample shows how to create and use KeyVaultClient:

### Using the Key Vault Client

```
var kv = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(Util.GetToken));
```

Once the client is created, you can invoke any of the get methods to get the necessary keys/secrets. Sample code as follows:

```
var sec = await kv.GetSecretAsync(<SECRET_IDENTIFIER>);
```

 **Note:** You can store a key in Key Vault by adding this information from the portal. The Azure Key Vault service accepts secret data, encrypts and stores it, returning a secret identifier, that may be used to retrieve the secret at a later time.

# Lab: Hosting and Deployment

## Scenario

In this lab, the students will first deploy a web application to Microsoft Azure. The application will manage an aquarium. It will store and display details of various fish and an image for each fish. The user can add/edit/delete a fish. In the second part of the lab, students will create an Azure storage container. They will then modify the application to store the image of the fish in Azure storage container instead of storing it locally.

## Objectives

After completing this lab, you will be able to:

- Create an App Service in Microsoft Azure.
- Deploy an application to Azure.
- Work with Azure SQL Database.
- Create a storage container on Azure.
- Upload images to storage containers from an application.

## Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD14\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD14_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D\\_MOD14\\_LAK.md](https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD14_LAK.md).

## Exercise 1: Deploying a Web Application to Microsoft Azure

### Scenario

In this exercise, you will first build and run a simple web application locally. For this, you will run migrations for your local database. You will then create an Azure App Service and Azure SQL database in Azure and configure your application to use the Azure SQL database. You will populate the database by using Migrate. Finally, you will deploy your application to Azure.

The main tasks for this exercise are as follows:

- Explore and run the application locally.
- Create a new Web App in Azure.
- Prepare the application for deployment.
- Deploy the application.
- Update the application and deploy in Azure.

## Exercise 2: Upload an Image to Azure Blob Storage

### Scenario

In this exercise, you will create an Azure storage account in Azure and a container in the storage account. You will then modify the web application you created to start storing the images in Azure Blob storage. The images will also be displayed from the storage by using the URL generated for each image. Also, you will change the code to upload images to the container. Finally, you will deploy your application to Azure.

The main tasks for this exercise are as follows:

- Create a blob storage account.
- Prepare the application for working with Azure Storage.
- Write the code to upload an image.
- Deploy and run the application in Azure.

**Question:** What are the advantages of deploying to Azure?

**Question:** When would you use Azure Blob storage?

## Module Review and Takeaways

In this module, you have learned about the importance of choosing the optimal means of making your ASP.NET Core MVC application accessible to your target audience. Whether by using self-hosting solutions or by using Azure services you can easily control accessibility to your application. Now that you have a handle on how to create and serve ASP.NET Core MVC Applications, you can fully create end to end applications. By using this course as a basis, you can further expand your knowledge, allowing you to create better ASP.NET Core applications.

You also learned about Azure and the options available to deploy your application on Azure. You learned about the various service available on Azure such as Azure Storage and Azure SQL that will help you build scalable applications. You also learned about various services available for monitoring and debugging applications deployed on Azure.

### Review Question

**Question:** How can you set an ASP.NET Core MVC application to dynamically use XML files which are added after the application is started?

### Best Practice

There is no single correct solution in regards to hosting and deployment. Every single case will need to be handled in a way that is most appropriate for the application requirements. Always analyze your requirements before choosing solutions.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
After running <b>dotnet publish</b> , you cannot find the published files in the project folder.	
After deploying the application to Azure App Service, you see the below in logs: <code>System.ArgumentException: The directory name 'D:\home\site\wwwroot\node_modules\' does not exist.</code>	
While uploading an image to Azure Storage from the application, you see an error.	

## Course Evaluation

Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

- Your evaluation of this course will help Microsoft understand the quality of your learning experience.
- Please work with your training provider to access the course evaluation form.
- Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.



---

## **Notes**

---

## **Notes**