



WEB PROGRAMMING ASP.NET MVC CORE

© 2017, NOEL LOPES

PROGRAM



- ASP.NET Core MVC introduction
- The Model-View Controller (MVC) pattern
- Creating ASP.NET Core MVC applications
- Introduction to C# and Razor
- Entity Framework Core
- Validating data
- Layouts and Navigation
- Security
- Authentication and Authorization
- ASP.NET Core MVC Tests Automation

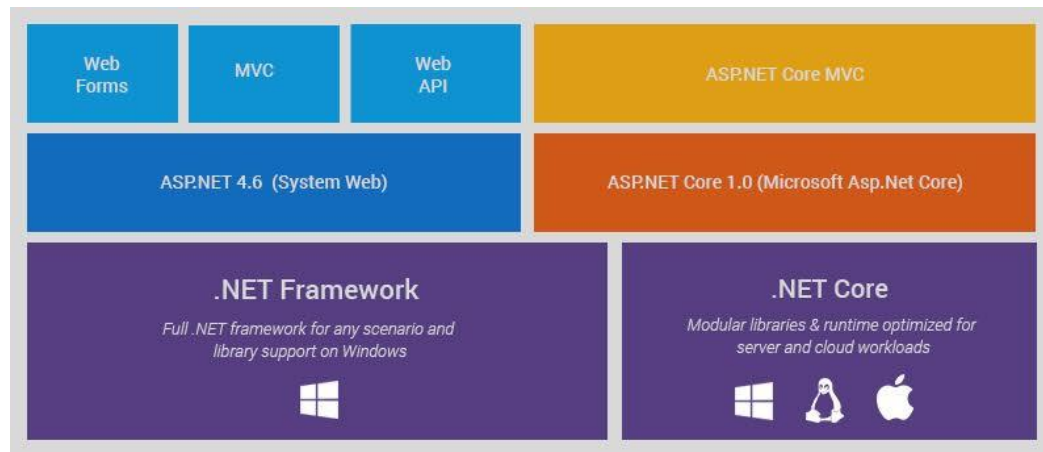
BIBLIOGRAPHY AND RESOURCES

- Adam Freeman, “Pro ASP.NET Core MVC”, 6th edition, Apress, 2016
- <https://www.asp.net/>

ASP.NET CORE MVC

- ASP.NET Core MVC is a web application development framework from Microsoft that combines the effectiveness and tidiness of model-view-controller (MVC) architecture, ideas and techniques from agile development, and the best parts of the .NET platform.
- It emphasizes clean architecture, design patterns, and testability, and it doesn't try to conceal how the Web works.

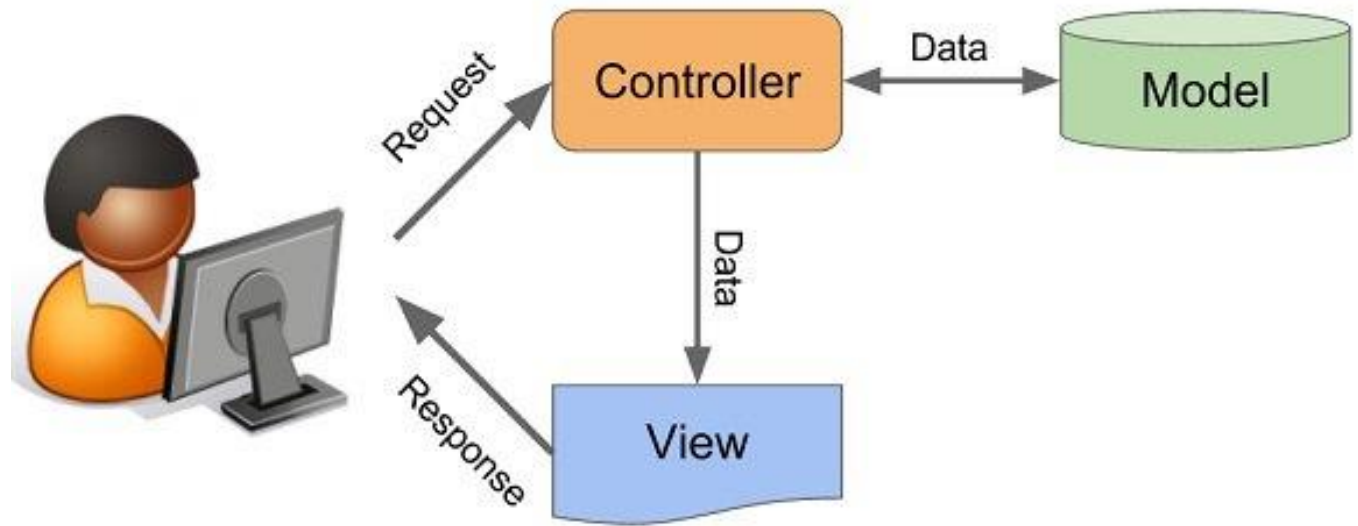
ASP.NET CORE MVC



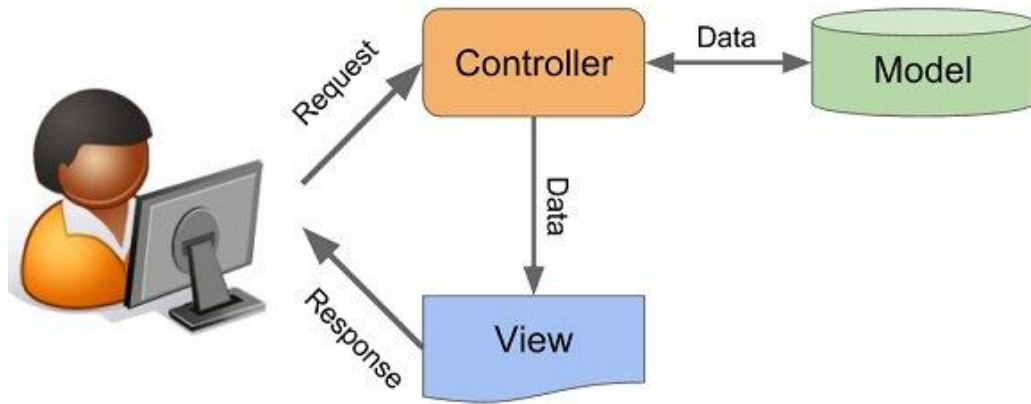
- ASP.NET Core is built on .NET Core, which is a cross-platform version of the .NET Framework without Windows-specific APIs.
- Web applications are increasingly hosted in small and simple containers in cloud platforms, and by embracing a cross-platform approach Microsoft extended the reach of .NET, making possible the deployment of ASP.NET Core applications to a broader set of hosting environments, and, as a bonus, made it possible for developers to create ASP.NET Core web applications on Linux and OS X.

MVC PATTERN

- ASP.NET Core MVC follows a pattern called model-view-controller (MVC), which guides the shape of an ASP.NET web application and the interactions between the components it contains.

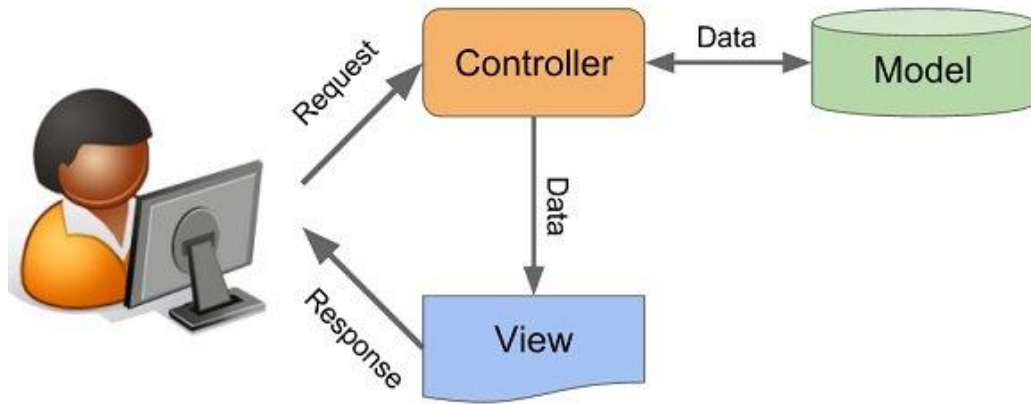


MVC PATTERN



- The MVC pattern dates back to 1978 and the Smalltalk project at Xerox PARC. But it has gained popularity recently as a pattern for web applications, for the following reasons:
 - User interaction with an application that adheres to the MVC pattern follows a natural cycle: the user takes an action, and in response the application changes its data model and delivers an updated view to the user. And then the cycle repeats.
 - This is a convenient fit for web applications delivered as a series of HTTP requests and responses.
 - Web applications necessitate combining several technologies (databases, HTML, and executable code, for example), usually split into a set of tiers or layers. The patterns that arise from these combinations map naturally onto the concepts in the MVC pattern.

MVC PATTERN



- The MVC architectural pattern helps to achieve separation of concerns, by separating an application into three main groups of components: Models, Views, and Controllers.
- Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries.
- The Controller chooses the View to display to the user, and provides it with any Model data it requires.

01

The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

02

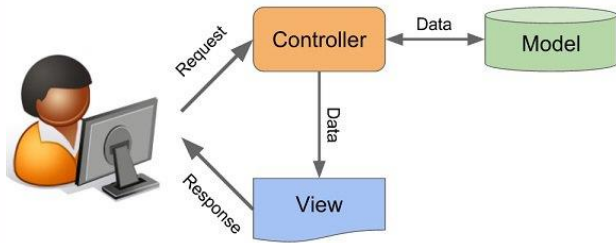
The pattern specifies where each kind of logic should be located in the app:

- The UI logic belongs in the view.
- Input logic belongs in the controller.
- Business logic belongs in the model.

03

This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another.

MVC PATTERN



MODEL RESPONSIBILITIES

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it.

Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application.

Strongly-typed views will typically use ViewModel types specifically designed to contain the data to display on that view; the controller will create and populate these ViewModel instances from the model.

VIEWMODEL

In an MVC web application, a ViewModel is a type that includes just the data a View requires for display (and perhaps for sending back to the server).

ViewModel types can also simplify model binding in ASP.NET MVC. ViewModel types are generally just data containers; any logic they may have should be specific to helping the View render data.

1

Views are responsible for presenting content through the user interface.

2

They use the Razor view engine to embed .NET code in HTML markup.

3

There should be minimal logic within views, and any logic in them should relate to presenting content.

4

A view template should never perform business logic or interact with a database directly. Instead, a view template should work only with the data that's provided to it by the controller.

VIEW RESPONSIBILITIES

CONTROLLER RESPONSIBILITIES

- Controllers handle user interaction, work with the model, and ultimately select a view to render.
- Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser.
- In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database.
- The controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name – it controls how the app responds to a given request).

01

Controllers should not be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, use the Single Responsibility Principle to push business logic out of the controller and into the domain model.

02

If you find that your controller actions frequently perform the same kinds of actions, you can follow the Don't Repeat Yourself principle by moving these common actions into filters.

CONTROLLERS

MVC WEB FRAMEWORK

Model

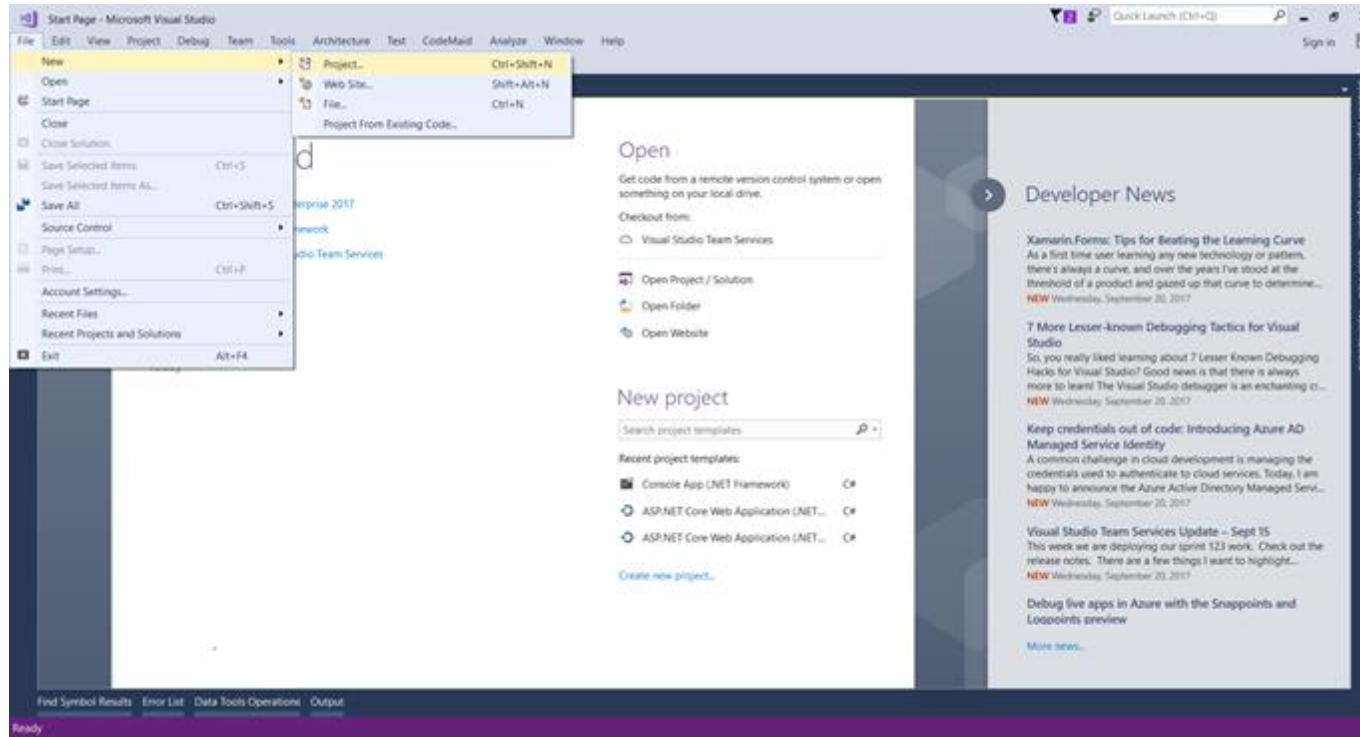
- Data Access Layer (e.g. using a tool like Entity Framework or Nhibernate)

View

- This is a template to dynamically generate HTML.

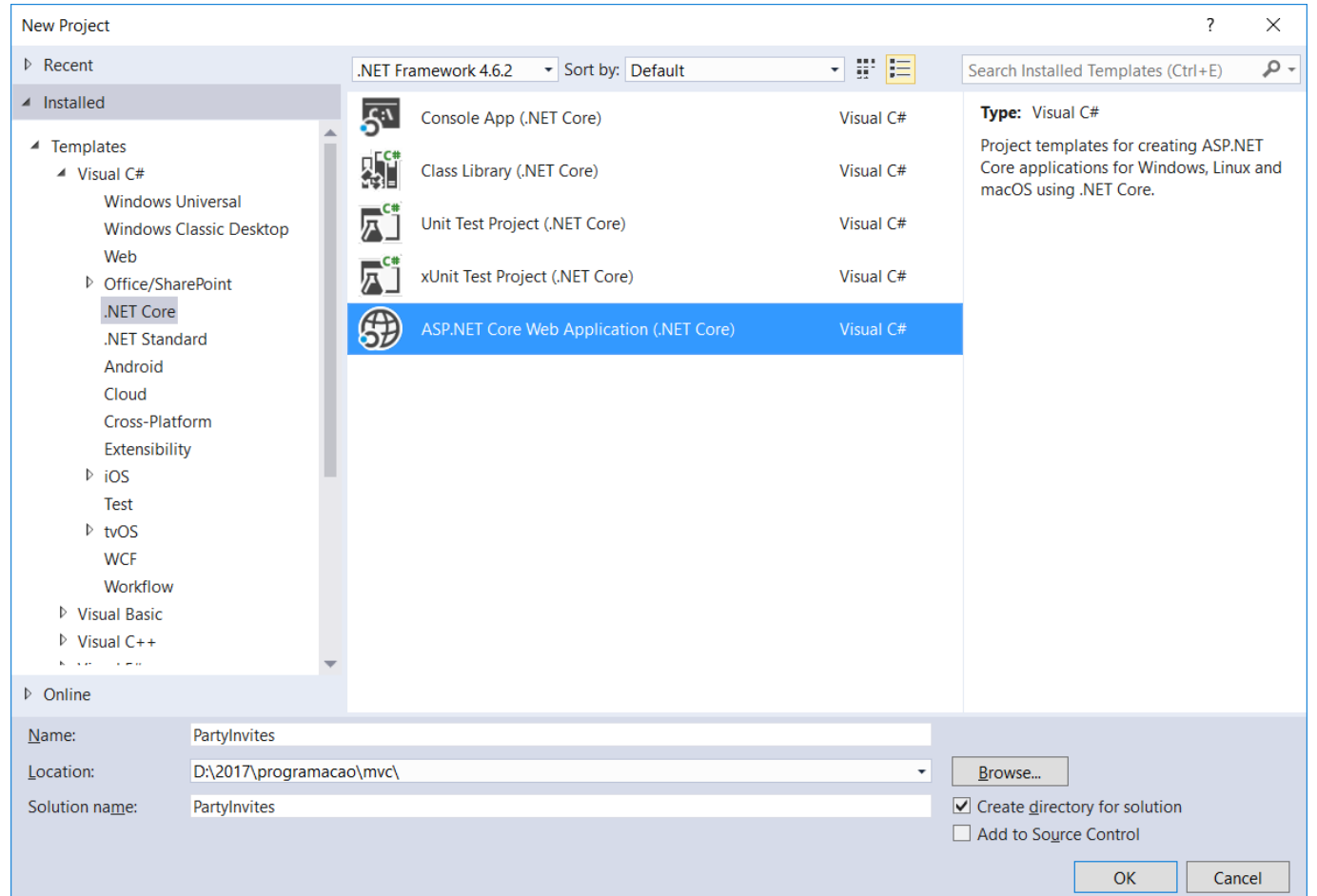
Controller

- Manages the relationship between the View and the Model. It responds to user input, talks to the Model, and decides which view to render

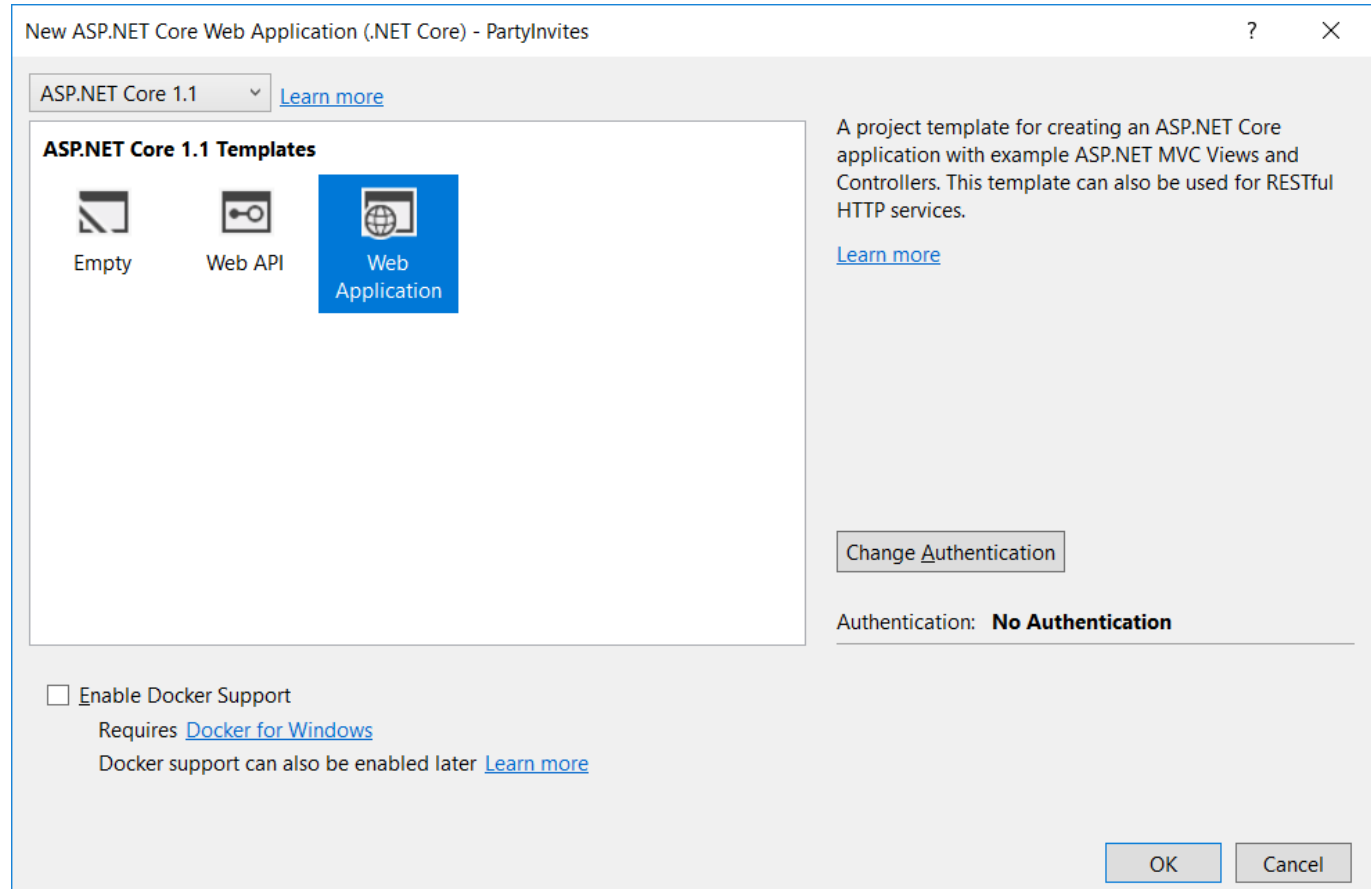


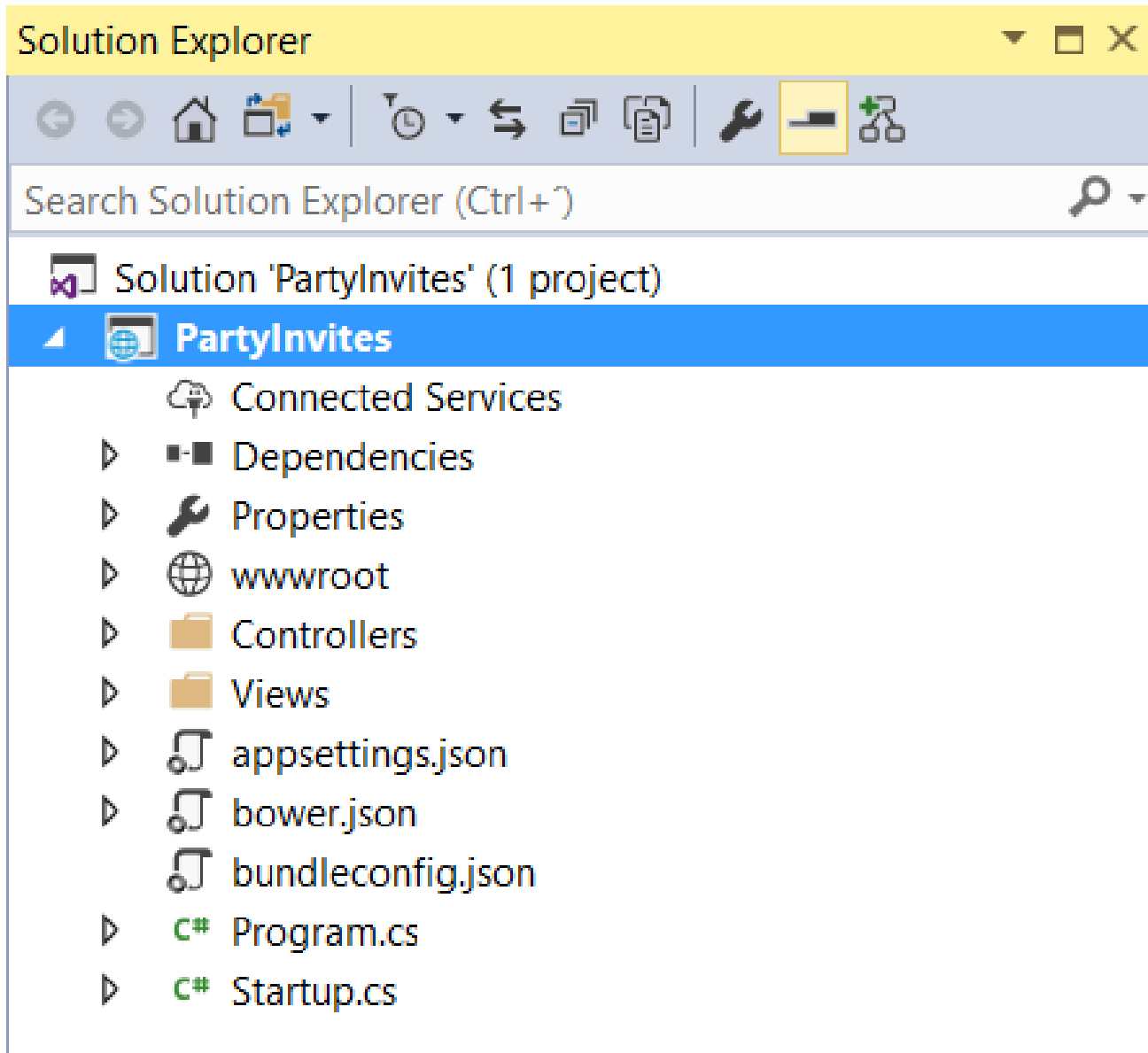
CREATE A NEW ASP.NET CORE MVC PROJECT USING VISUAL STUDIO

CREATE A NEW ASP.NET CORE MVC PROJECT USING VISUAL STUDIO



CREATE A NEW ASP.NET CORE MVC PROJECT USING VISUAL STUDIO





SOLUTION EXPLORER

MVC APPLICATION STRUCTURE

DIRECTORY	PURPOSE
/Models	Data (business) objects
/Views	UI template files for rendering output (e.g. HTML)
/Controllers	Controller classes that handle URL requests
/Views/Shared	Layouts and views that are not specific to a single controller.
/Views/_ViewImports.cshtml	Contains the namespaces that will be included in Razor view files.
/Views/_ViewStart.cshtml	Default layout for the Razor view engine.
/wwwroot	Static content (e.g. CSS, image, javascript files)
/wwwroot/lib	Third-party JavaScript and CSS packages

MVC APPLICATION STRUCTURE

DIRECTORY	PURPOSE
/Areas	Areas are a way of partitioning a large application into smaller pieces.
/Dependencies	Provides details of all the packages a project relies on.
/Components	This is where view component classes, which are used to display self-contained features such as shopping carts, are defined.
/Data	This is where database context classes are defined
/Migrations	This is where details of database schemas are stored so that databases can be updated.
/bower.json	List of packages managed by the Bower package manager. This file is hidden by default.
/project.json	Basic configuration options for the project, including the NuGet packages it uses
/Program.cs	This class configures the hosting platform for the application
/Startup.cs	This class configures the application

RUNNING THE APPLICATION

ASP.NET Core

Windows

Linux

OSX

Learn how to build ASP.NET apps that can run anywhere.

[Learn More](#)

Application uses

- Sample pages using ASP.NET Core MVC
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

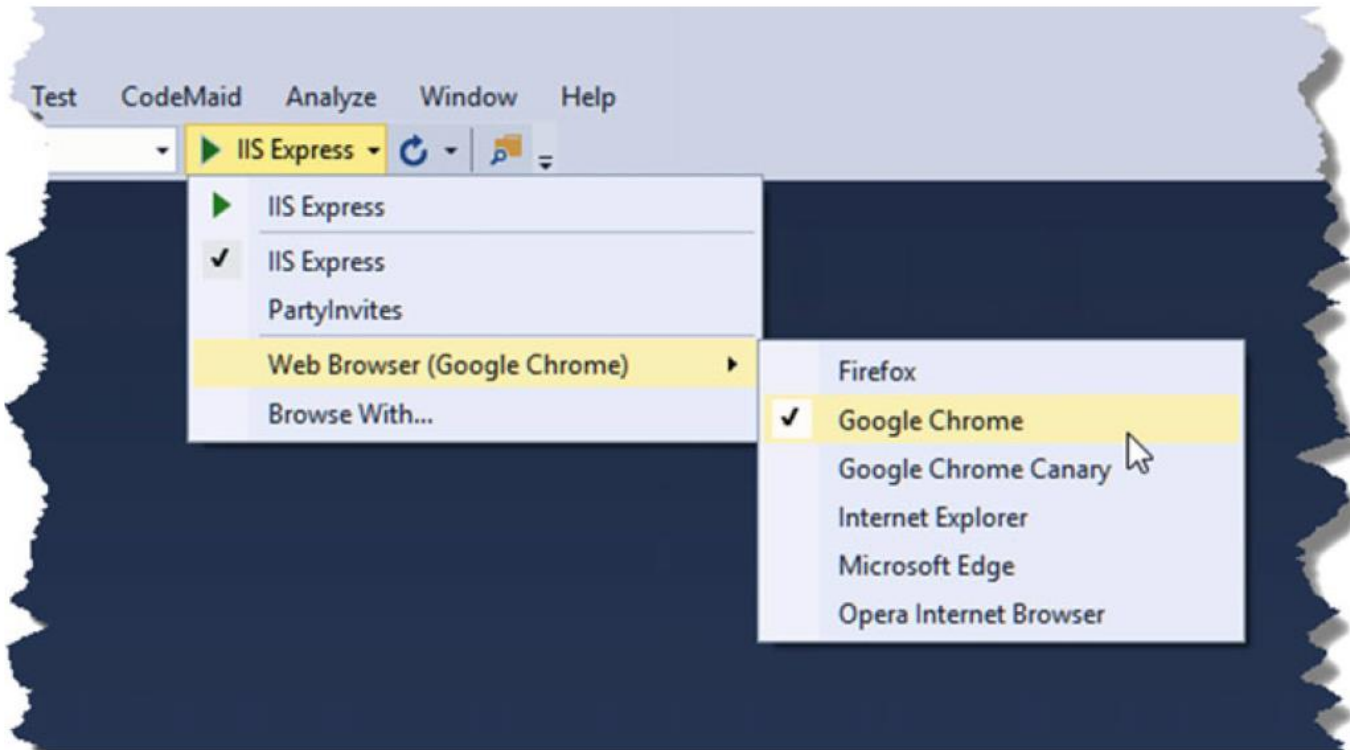
- [Add a Controller and View](#)
- [Manage User Secrets using Secret Manager.](#)
- [Use logging to log a message.](#)
- [Add packages using NuGet.](#)
- [Add client packages using Bower.](#)
- [Target development, staging or production environment.](#)

Overview

- [Conceptual overview of what is ASP.NET Core](#)
- [Fundamentals of ASP.NET Core such as Startup and middleware.](#)
- [Working with Data](#)
- [Security](#)
- [Client side development](#)
- [Develop on different platforms](#)
- [Read more on the documentation site](#)

Run & Deploy

- [Run your app](#)
- [Run tools such as EF migrations and more](#)
- [Publish to Microsoft Azure Web Apps](#)



CHANGING THE BROWSER

CONTROLLERS

Incoming requests are handled by controllers . In ASP.NET Core MVC, controllers are just C# classes (usually inheriting from the `Microsoft.AspNetCore.Mvc.Controller` class, which is the built-in MVC controller base class).

Controller classes handle communication from the user, overall application flow, and application-specific logic. Every public method in a controller is callable as an HTTP endpoint. The methods within a controller are called controller actions. Their job is to respond to URL requests perform the appropriate actions, and return a response back to the browser or user that invoked the URL.

The MVC convention is to put controllers in the Controllers folder, which Visual Studio created when it set up the project.

CONVENTION OVER CONFIGURATION

“We know, by now, how to build a web application. Let’s roll that experience into the framework so we don’t have to configure absolutely everything again.”

Each controller’s class name ends with Controller (e.g. ProductController, HomeController) in the Controllers directory.

Views that controllers use live in a subdirectory of the Views main directory and are named according to the controller name (e.g. the view for the ProductController is /Views/Product)

CONTROLLERS

```
public class HomeController : Controller {  
    public IActionResult Index() {  
        return View();  
    }  
  
    public IActionResult Contact() {  
        ViewData["Message"] = "Contacts ...";  
        return View();  
    }  
}
```

Solution Explorer

Search Solution Explorer (Ctrl+')

Solution 'PartyInvites' (1 project)

- PartyInvites
 - Connected Services
 - Dependencies
 - Properties
 - wwwroot

Controllers

- controllers
- controllers.settings.json
- controllers.json
- controllers.config.json
- controllers.css

Controller...

- New Item... Ctrl+Shift+A
- Existing Item... Shift+Alt+A
- New Scaffolded Item...
- New Folder
- Docker Support
- Class...

View in Browser (Google Chrome) Ctrl+Shift+W

Browse With...

Cleanup Selected Code

Collapse Recursively

Add

- Scope to This
- New Solution Explorer View
- Exclude From Project

Cut Ctrl+X

Copy Ctrl+C

Delete Del

Rename

Open Folder in File Explorer

Properties Alt+Enter

ADDING A CONTROLLER

Add MVC Dependencies

☐ Minimal Dependencies

Adds the minimal packages and references to start using ASP.NET MVC Core.

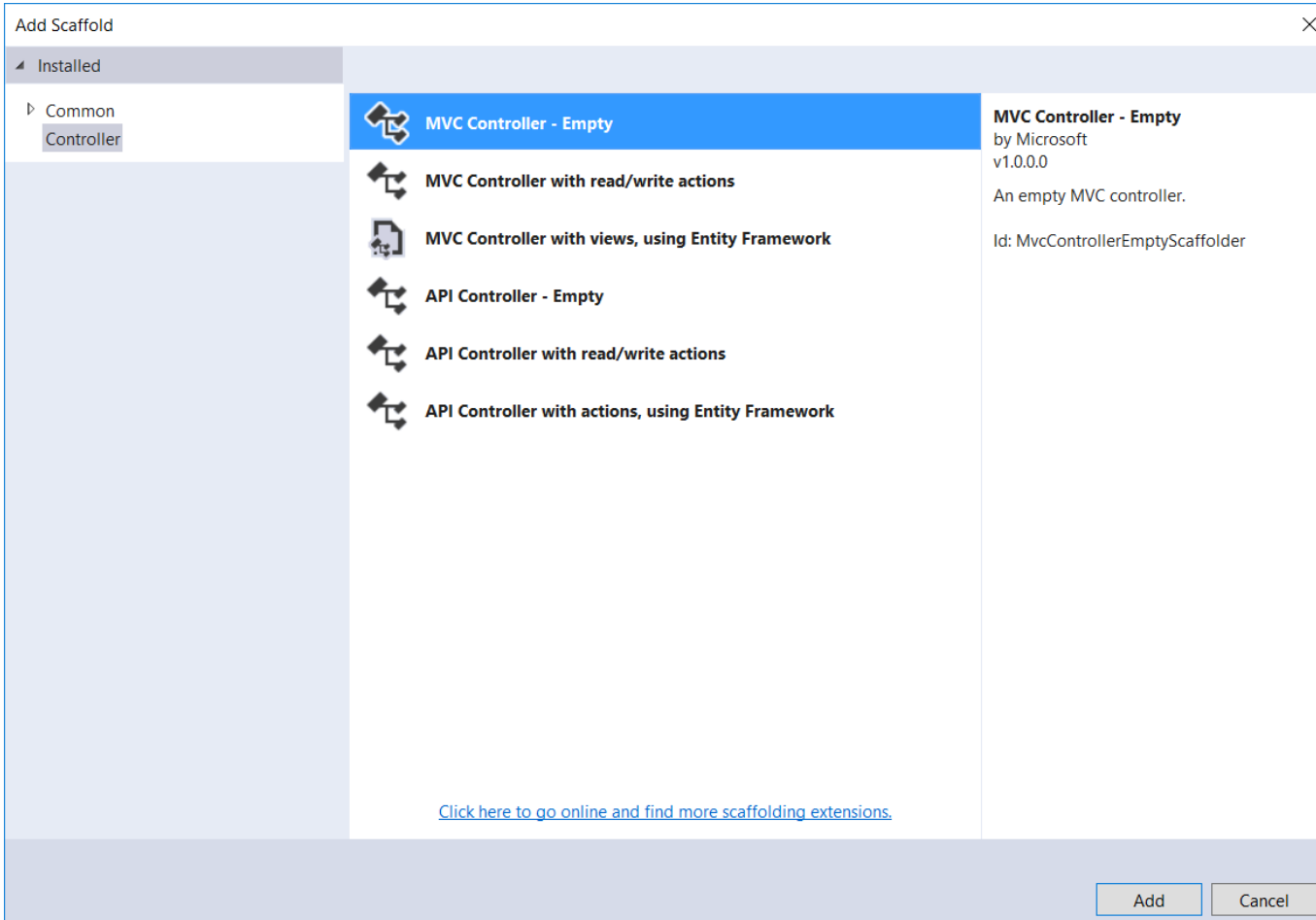
☒ Full Dependencies

Adds packages and configurations, as well as a default layout, error page, script libraries, and script bundling to your application.

Add

Cancel

ADDING A CONTROLLER



ADDING A CONTROLLER

Add Controller

Controller name:

HelloWorldController

Add

Cancel

ADDING A CONTROLLER

CONTROLLER ACTIONS

```
public class HelloWorldController : Controller {  
    //  
    // GET: /HelloWorld/  
    public string Index() {  
        return "This is my default action...";  
    }  
  
    //  
    // GET: /HelloWorld/Welcome/  
    public string Welcome() {  
        return "Welcome action method...";  
    }  
}
```

UNDERSTANDING ROUTES

- MVC applications use the ASP.NET *routing system*, which decides how URLs map to controllers and actions. A route is a rule that is used to decide how a request is handled.
- The format for routing is defined in the `Startup.cs` file.

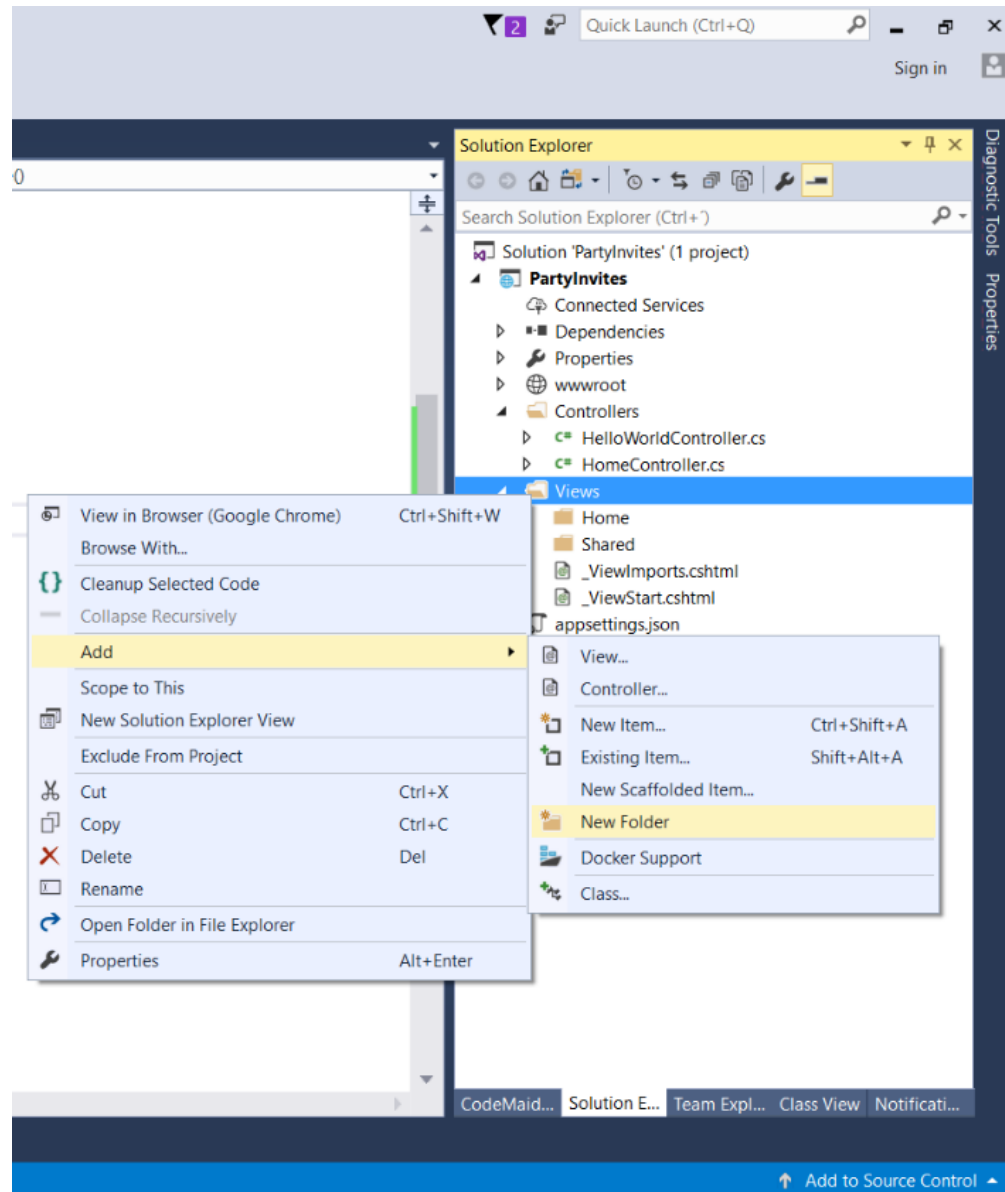
```
app.UseMvc(routes => {  
    routes.MapRoute(  
        name: "default",  
        template: "{controller=Home}/{action=Index}/{id?}");  
});
```


VIEWS

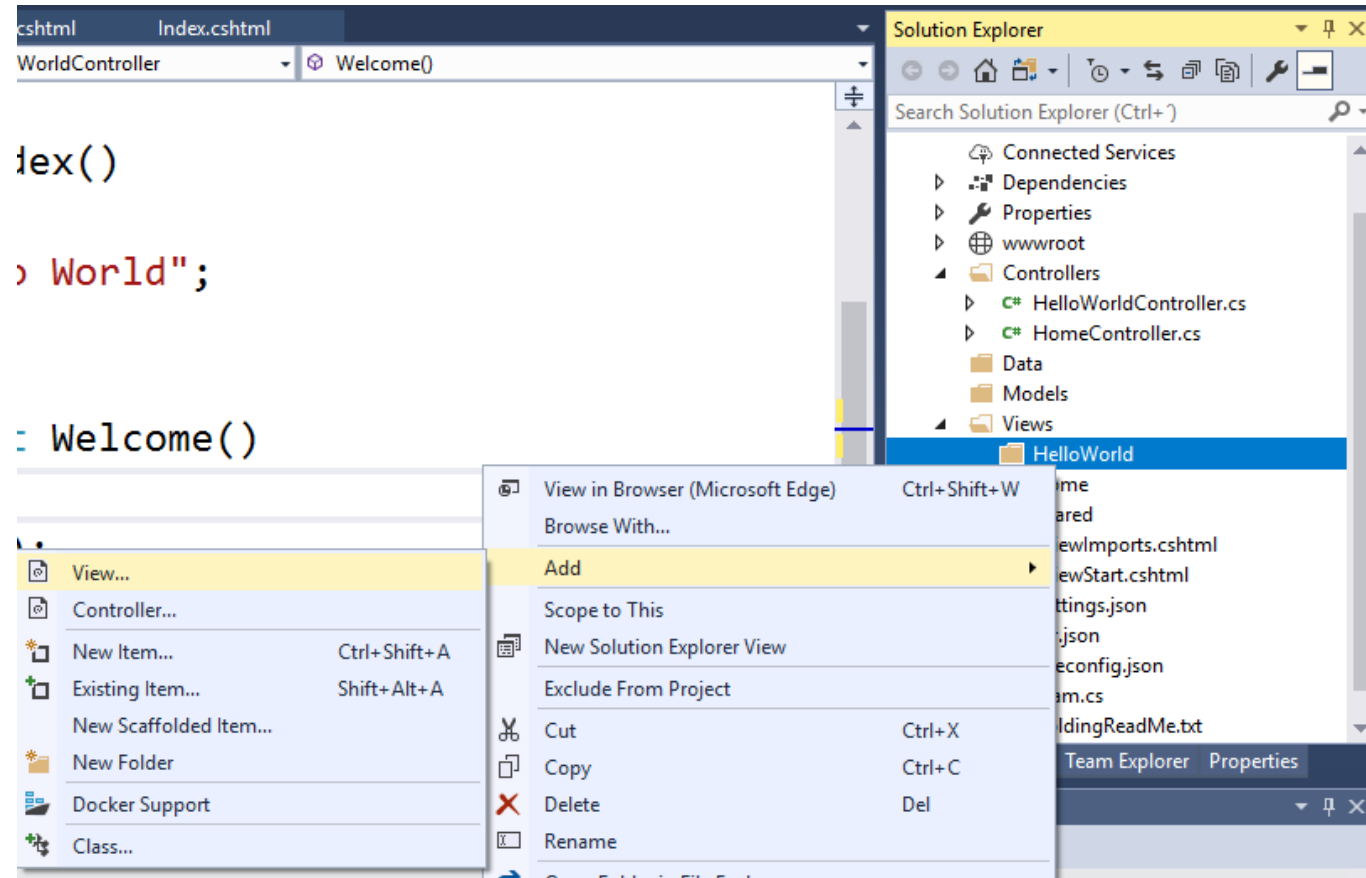
- The view is responsible for providing the user interface (UI) to the user. After the controller has executed the appropriate logic for the requested URL, it delegates the display to the view.
- Views are stored in the Views folder, organized into subfolders. Views that are associated with the Home controller, for example, are stored in a folder called Views/Home . Views that are not specific to a single controller are stored in a folder called Views/Shared .
- Visual Studio creates the Home and Shared folders automatically when the Web Application template is used and puts in some placeholder views to get the project started.

```
public ActionResult Welcome()  
{  
    return View();  
}
```

MODIFYING THE
CONTROLLER TO
RENDER A VIEW



ADDING A VIEW



CREATING A VIEW

Add View ✕

View name:

Template:

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

CREATING A VIEW

VIEWS

MVC uses the naming convention to find the View automatically. The convention is that the view has the name of the action method and is contained in a folder named after the controller.

```
@{  
    ViewData["Title"] = "Welcome";  
}
```

```
<h2>Welcome</h2>
```

```
<p>Hello from Welcome action (Hello World  
Controller)</p>
```

ACTION METHODS RETURN TYPES

- Besides strings and `ViewResult` objects action methods can return other results. For example, if the method returns a `RedirectResult`, the browser will be redirected to another URL. If it returns an `HttpUnauthorizedResult`, I force the user to log in. These objects are collectively known as action results. The action result system lets you encapsulate and reuse common responses in actions.

- The whole point of a web application platform is to construct and display *dynamic* output. In MVC, it is the controller's job to construct some data and pass it to the view, which is responsible for rendering it to HTML.
- One way to pass data from the controller to the view is by using the ViewBag object, which is a member of the Controller base class. ViewBag is a dynamic object to which you can assign arbitrary properties, making those values available in whatever view is subsequently rendered.

ADDING DYNAMIC OUTPUT

PASSING DATA FROM THE CONTROLLER TO THE VIEW

Data for the View is provided when the ViewBag.Message property is assigned a value.

The Message property didn't exist until the moment it was assigned a value. This allows to pass data from the controller to the view in a free and fluid manner, without having to define classes ahead of time.

```
public IActionResult Index() {  
    int hour = DateTime.Now.Hour;  
  
    string message;  
  
    if (hour >= 7 && hour < 12) {  
        message = "Good morning";  
    } else if (hour >= 12 && hour < 20) {  
        message = "Good afternoon";  
    } else {  
        message = "Good evening";  
    }  
  
    ViewBag.Message = message;  
  
    return View();  
}
```

```
@{
```

```
    ViewData["Title"] = "Home";
```

```
}
```

```
<h2>Home</h2>
```

```
<p>@ ViewBag.Message World !!!</p>
```

```
<p>We are going to have an exciting party.</p>
```

RETRIEVING A
VIEWBAG DATA
VALUE IN THE
VIEW

MODELS

- Models represent the domain the application focuses on. They describe the data as well as the business rules for how the data can be changed and manipulated.
- *The model* is the most important part of the application. It is a representation of the real-world objects, processes, and rules that define the subject, known as the *domain*, of the application. The model, often referred to as a *domain model*, contains the C# objects (known as *domain objects*) that make up the universe of the application and the methods that manipulate them.
- The views and controllers expose the domain to the clients in a consistent manner, and a well-designed MVC application starts with a well-designed model, which is then the focal point as controllers and views are added.
- The MVC convention is that the classes that make up a model are placed inside a folder called the Models.

```
namespace PartyInvites.Models {  
    public class GuestResponse {  
        public string Name { get; set; }  
        public string Phone { get; set; }  
        public string Email { get; set; }  
        public bool? WillAttend { get; set; }  
    }  
}
```

MODELS

```
namespace PartyInvites.Controllers {  
    public class HomeController : Controller {  
        // ...  
  
        public ActionResult Rsvp() {  
            return View();  
        }  
    }  
}
```

RÉPONDEZ S'IL
VOUS PLAÎT
(RSVP)
CONTROLLER

STRONGLY TYPED VIEW

A strongly typed view is intended
to render a specific model type

```
@model PartyInvites.Models.GuestResponse
```

```
@{  
    ViewData["Title"] = "Repondez S'il Vous Plait";  
}
```

```
<h2>Repondez S'il Vous Plait</h2>
```

```
@{
```

```
    ViewData["Title"] = "Home";
```

```
}
```

```
<h2>Home</h2>
```

```
<p>@ViewBag.Message World !!!</p>
```

```
<p>We are going to have an exciting party.</p>
```

```
<a asp-action="Rsvp">Repondez S'il Vous Plait</a>
```

LINKING ACTION METHODS

LINKING ACTION METHODS

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

<h2>Home</h2>

<p>@ViewBag.Message World !!!</p>

<p>We are going to have an exciting party.</p>

<a asp-action="Rsvp">Repondez S'il Vous Plait</a>
```

- The attribute `asp-action` is a *tag helper* attribute, which is an instruction for Razor that will be performed when the view is rendered.
- An `href` attribute to the `a` element that contains a URL for an action method will be rendered.
- There is an important principle at work here, which is that you should use the features provided by MVC to generate URLs, rather than hard-code them into your views. When the tag helper created the `href` attribute for the `a` element, it inspected the configuration of the application to figure out what the URL should be. This allows the configuration of the application to be changed to support different URL formats without needing to update any views.

CREATING A FORM VIEW

The `asp-action` attribute uses the application's URL routing configuration to set the action attribute to a URL that will target a specific action method.

```
<form asp-action="Rsvp" method="post">  
    <!-- ... -->  
</form>
```

CREATING A FORM VIEW

Each element is associated with the model property using the `asp-for` attribute, which is another tag helper attribute.

The `asp-for` attribute on the `label` element sets the value of the `for` attribute.

The `asp-for` attribute on the `input` element sets the `id` and `name` elements.

```
<form asp-action="Rsvp" method="post">
  <table>
    <tr>
      <td>
        <label asp-for="Name">Name:</label>
      </td>
      <td>
        <input asp-for="Name" />
      </td>
    </tr>

    <!-- ... -->

  </table>
</form>
```

CREATING A FORM VIEW

```
<form asp-action="Rsvp" method="post">
  <table>
    <!-- ... -->

    <tr>
      <td>
        <label asp-for="WillAttend">I will</label>
      </td>
      <td>
        <select asp-for="WillAttend">
          <option value="">Choose an option</option>
          <option value="true">attend the party</option>
          <option value="false">not go the party</option>
        </select>
      </td>
    </tr>

    <!-- ... -->
  </table>
</form>
```

CREATING A FORM VIEW

```
<form asp-action="Rsvp" method="post">
  <table>
    <!-- ... -->
    <tr>
      <td>
        <button type="submit">
          Submit response
        </button>
      </td>
      <td></td>
    </tr>
  </table>
</form>
```

GET AND POST REQUESTS

Web applications generally use GET requests for reads and POST requests for writes (which typically include updates, creates, and deletes).

A GET request represents an independent read-only operation. You can send a GET request to a server repeatedly with no ill effects, because a GET should not change state on the server. Moreover, you can bookmark the GET request because all the parameters are in the URL (thus the form input values are preserved). A GET request is what a browser issues normally each time someone clicks a link.

Performing a create, delete or edit operation in response to a GET request (or for that matter, any other operation that changes data) opens up a security hole.

A POST request generally modifies state on the server, and repeating the request might produce undesirable effects (*e.g.* double billing).

```
public class HomeController : Controller {  
  
    // ...  
  
    [HttpGet]  
    public ActionResult Rsvp() {  
        return View();  
    }  
  
    [HttpPost]  
    public ActionResult Rsvp(GuestResponse response) {  
        //TODO: Store guest response  
        return View();  
    }  
  
    // ...  
}
```

GET VS POST

MODEL BINDING

- *Model binding* is a useful MVC feature whereby incoming data is parsed and the key/value pairs in the HTTP request are used to populate properties of domain model types. It eliminates the grind and toil of dealing with HTTP requests directly and lets you work with C# objects rather than dealing with individual data values sent by the browser.
- Model binding free us from the tedious and error-prone task of having to inspect an HTTP request and extract all the data values that are required.

ACADEMIC EXAMPLE

For now we will store data in an in-memory collection of objects. This isn't useful in a real application because the response data will be lost when the application is stopped or restarted

```
// NEVER DO THIS !!!  
// This is only for demonstration/academic purposes  
  
public class Repository {  
    private static List<GuestResponse> responses =  
        new List<GuestResponse>();  
  
    public static IEnumerable<GuestResponse> Responses {  
        get {  
            return responses;  
        }  
    }  
  
    public static void AddResponse(GuestResponse response) {  
        responses.Add(response);  
    }  
}
```



```
namespace PartyInvites.Controllers {  
    public class HomeController : Controller {  
        // ...  
  
        [HttpPost]  
        public ActionResult Rsvp(GuestResponse response) {  
            Repository.AddResponse(response);  
  
            return View("Thanks", response);  
        }  
  
        // ...  
    }  
}
```

RSVP POST
ACTION

The Thanks.cshtml view uses Razor to display content based on the value of the GuestResponse properties that I passed to the View method in the Rsvp action method.

The Razor `@model` expression specifies the domain model type with which the view is strongly typed.

To access the value of a property in the domain object, use `Model.PropertyName`. For example, to get the value of the Name property, call `Model.Name`.

```
@model PartyInvites.Models.GuestResponse
```

```
@{  
    ViewData["Title"] = "Thanks";  
}
```

```
<h2>Thank you, @ Model.Name !!!</h2>
```

```
@ if (Model.WillAttend == null) {  
    @: Thank you for your answer. When you decide if you  
    can come to the party, give me a call.  
} else if (Model.WillAttend == true) {  
    @: Thank you for your answer. When you decide if you  
    can come to the party, give me a call.  
} else { // Model.WillAttend == false  
    @: Sorry to hear that you can't make it, but tanks  
    for letting us know.  
}
```

```
namespace PartyInvites.Controllers {  
    public class HomeController : Controller {  
  
        // ...  
  
        public ActionResult GuestList() {  
            return View(Repository.Responses);  
        }  
  
        // ...  
    }  
}
```

GUESTLIST ACTION

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
```

```
@{  
    ViewData["Title"] = "GuestList";  
}
```

```
<h2>Guest List</h2>
```

```
<table>  
    <thead>  
        <tr>  
            <td>Name</td>  
            <td>Phone</td>  
            <td>Email</td>  
            <td>Will Attend</td>  
        </tr>  
    </thead>  
    <tbody>  
        <!-- ... -->  
    </tbody>  
</table>
```

GUESTLIST VIEW

```
foreach (var r in Model) {  
    <tr>  
        <td>@r.Name</td>  
        <td>@r.Phone</td>  
        <td>@r.Email</td>  
        <td>  
            @if (r.WillAttend == true) {  
                @: Yes  
            } else if (r.WillAttend == false) {  
                @: No  
            } else {  
                @: Don't know  
            }  
        </td>  
    </tr>  
}
```

GUESTLIST VIEW

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
```

```
@{  
    ViewData["Title"] = "GuestList";  
}
```

```
<h2>Guest List</h2>
```

```
@if (Model.Count() == 0) {  
    <p>  
        Nobody has yet confirm that s/he will come to  
the party. Be the first to do that and I will offer  
you a special drink.  
    </p>  
    <a asp-action="Rsvp">Repondez S'il Vous Plait</a>  
} else {  
    <!-- ... -->  
}
```

GUESTLIST VIEW

GUESTLIST CONTROLLER

```
namespace PartyInvites.Controllers {  
    public class HomeController : Controller {  
  
        // ...  
  
        public ActionResult GuestList() {  
            return View(Repository.Responses.Where(r => r.WillAttend == true));  
        }  
  
        // ...  
    }  
}
```

ADDING VALIDATIONS

01

Without validation, users could enter nonsense data or even submit an empty form. In an MVC application, you will typically apply validation to the domain model rather than in the user interface. This means that you define validation in one place, but it takes effect anywhere in the application that the model class is used.

02

MVC supports declarative validation rules defined with attributes from the `System.ComponentModel.DataAnnotations` namespace, meaning that validation constraints are expressed using the standard C# attribute features.

03

MVC automatically detects the attributes and uses them to validate data during the model-binding process.

ADDING VALIDATIONS

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {
    public class GuestResponse {
        [Required(ErrorMessage = "Please enter your name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter your phone number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please enter your email address")]
        [RegularExpression(".*+\\@.*+\\.+.",
            ErrorMessage = "Please enter a valid email address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please specify whether you'll attend")]
        public bool? WillAttend { get; set; }
    }
}
```

ADDING VALIDATION

If the `ModelState.IsValid` property returns false, then I know that there are validation errors. The object returned by the `ModelState` property provides details of each problem that has been encountered, but we don't need to get into that level of detail, because we can rely on a useful feature that automates the process of asking the user to address any problems by calling the `View` method without any parameters.

```
[HttpPost]
public ActionResult Rsvp(GuestResponse response) {
    if (ModelState.IsValid) {
        Repository.AddResponse(response);
        return View("Thanks", response);
    } else {
        // There are validation errors
        return View();
    }
}
```

ADDING VALIDATION

When MVC renders a view, Razor has access to the details of any validation errors associated with the request, and tag helpers can access the details to display validation errors to the user.

The `asp-validation-summary` attribute is applied to a `div` element, and it displays a list of validation errors when the view is rendered.

```
<form asp-action="Rsvp" method="post">
    <!-- ... -->

    <div asp-validation-summary="All"></div>
</form>
```

HIGHLIGHTING INVALID FIELDS

- When there are invalid fields, the data entered is preserved and displayed again. This is another benefit of model binding, and it simplifies working with form data.

```
.field-validation-error {  
    color: #f00;  
}  
  
.field-validation-valid {  
    display: none;  
}  
  
.input-validation-error {  
    border: 1px solid #f00;  
    background-color: #fee;  
}  
  
.validation-summary-errors {  
    font-weight: bold;  
    color: #f00;  
}  
  
.validation-summary-valid {  
    display: none;  
}
```

BOOTSTRAP

Bootstrap, is CSS framework originally developed by Twitter that has become a major open source project in its own right and which has become a mainstay of web application development.

The convention is that Bootstrap and other third-party CSS and JavaScript packages are installed into the `wwwroot/lib` folder

```
@{
```

```
    ViewData["Title"] = "Home";
```

```
}
```

```
<h2>Home</h2>
```

```
<p>@ViewBag.Message World !!!</p>
```

```
<p>We are going to have an exciting party.</p>
```

```
<p>Are you coming to the party?</p>
```

```
<a asp-action="Rsvp" class="btn btn-primary">
```

```
    Submit answer
```

```
</a>
```

```
<a asp-action="GuestList" class="btn btn-info">
```

```
    See who is coming to the party
```

```
</a>
```

BOOTSTRAP

Bootstrap defines classes that can be used to style forms.

<https://bootstrapcreative.com/resources/bootstrap-3-css-classes-index/>

```
<div class="panel panel-primary" style="margin-top:2ex;">
  <div class="panel-heading text-center">
    <h2>Repondez S'il Vous Plait</h2>
  </div>

  <form asp-action="Rsvp" method="post">
    <div class="panel-body">
      <!-- ... -->

      <div asp-validation-summary="All"></div>
    </div>
    <div class="panel-footer">
      <button type="submit" class="btn btn-primary">
        Submit response
      </button>
    </div>
  </form>
</div>
```

BOOTSTRAP

The form-group class is used to style the element that contains the label and the associated input or select element.

```
<div class="form-group">  
  <label asp-for="Name">Name:</label>  
  <input asp-for="Name" class="form-control" />  
</div>
```

```
<div class="form-group">  
  <label asp-for="Phone">Phone:</label>  
  <input asp-for="Phone" class="form-control" />  
</div>
```

```
<div class="form-group">  
  <label asp-for="Email">Email:</label>  
  <input asp-for="Email" class="form-control" />  
</div>
```

```
<!-- ... -->
```

BOOTSTRAP

TABLES

```
<table class="table table-striped">
  <thead>
    <tr>
      <th>Name</th>
      <th>Phone</th>
      <th>Email</th>
      <th>Will attend</th>
    </tr>
  </thead>

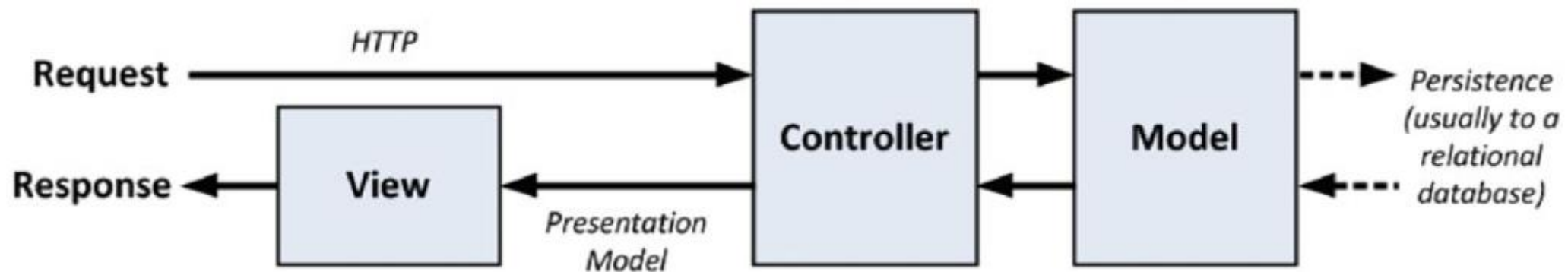
  <!-- ... -->
</table>
```


USING A DIFFERENT BOOTSTRAP THEME

- Download and rename the CSS files from <https://bootswatch.com/flatly/>
- Add the files to the CSS folder
- Change the Shared/_Layout.cshtml file

```
<environment names="Development">
    <link rel="stylesheet" href="~/css/flatly_bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet" href="~/css/flatly_bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

THE ASP.NET IMPLEMENTATION OF MVC



MODELS

View models

Represent just data passed from the controller to the view

Domain models

Contain the data in a business domain, along with the operations, transformations, and rules for creating, storing, and manipulating that data, collectively referred to as the model logic.

DOMAIN MODELS

A model should

- Contain the domain data
- Contain the logic for creating, managing, and modifying the domain data
- Provide a clean API that exposes the model data and operations on it

A model should not

- Expose details of how the model data is obtained or managed (in other words, details of the data storage mechanism should not be exposed to controllers and views)
- Contain logic that transforms the model based on user interaction (because that is the controller's job)
- Contain logic for displaying data to the user (that is the view's job)

CONTROLLERS

A controller should

- Contain the actions required to update the model based on user interaction

The controller should not

- Contain logic that manages the appearance of data (that is the job of the view)
- Contain logic that manages the persistence of data (that is the job of the model)

VIEWS

Views should

- Contain the logic and markup required to present data to the user

Views should not

- Contain complex logic (this is better placed in a controller)
- Contain logic that creates, stores, or manipulates the domain model



1

Razor is the view engine responsible for incorporating data into HTML documents.



2

Razor provides features that make it easy to work with the rest of the ASP.NET Core MVC using C# statements.



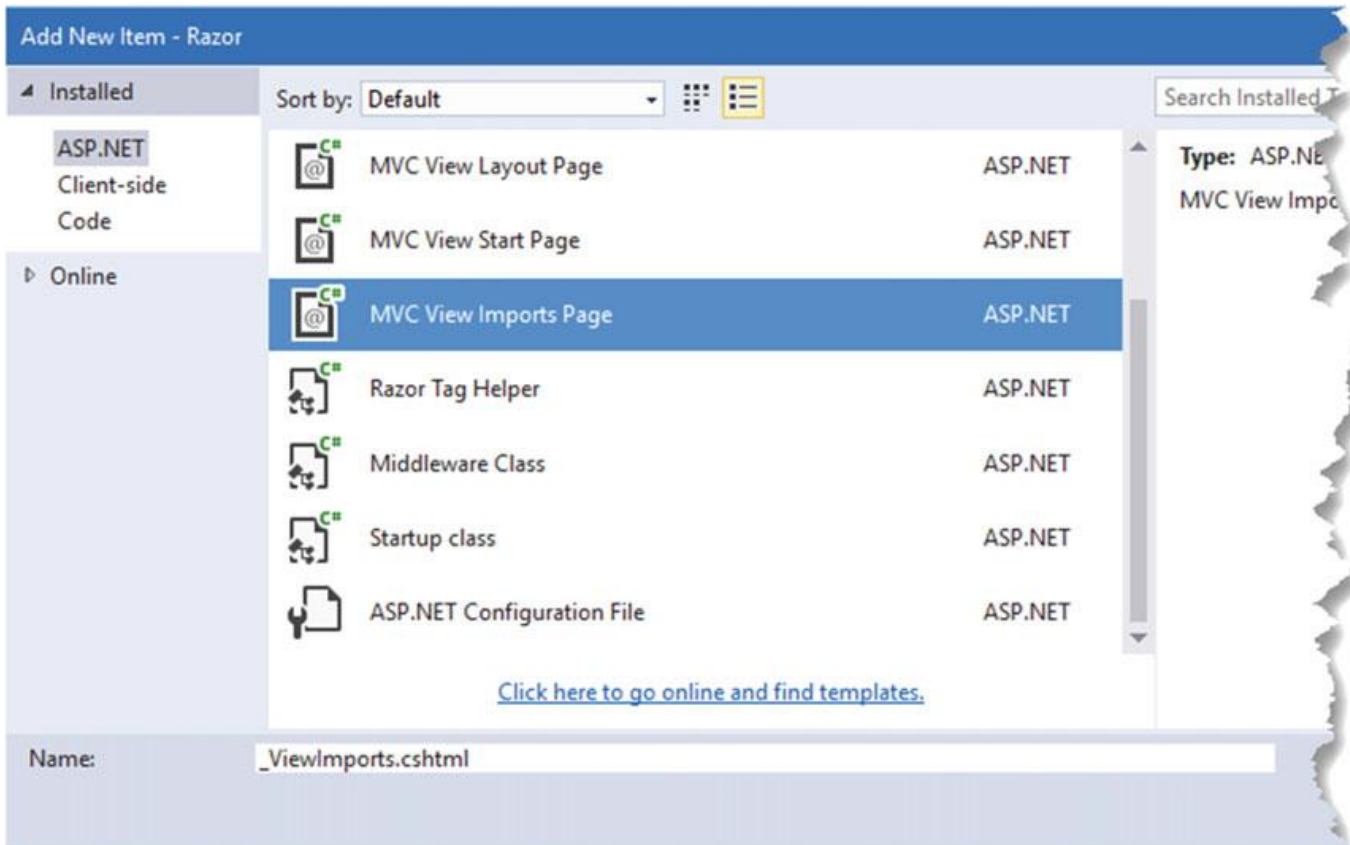
3

Razor expressions are added to static HTML in view files. The expressions are evaluated to generate responses to client requests.

RAZOR

VIEW IMPORTS

- By default, all types that are referenced in a strongly typed Razor view must be qualified with a namespace (e.g. `@model PartyInvites.Models.GuestResponse`)
- Alternatively, you can specify a set of namespaces that should be searched for types by adding a view imports file to the project. The view imports file is placed in the Views folder and is named `_ViewImports.cshtml`.
- You can also add an `@using` expression to individual view files, which allows types to be used without namespaces in a single view.



CREATE A VIEW
IMPORTS PAGE

WORKING WITH LAYOUTS

A real project can have dozens of views, and some views will have shared content. Duplicating shared content in views becomes hard to manage, especially when you need to make a change and have to track down all of the views that need to be altered. A better approach is to use a Razor layout, which is a template that contains common content and that can be applied to one or more views. When you make a change to a layout, the change will automatically affect all the views that use it.

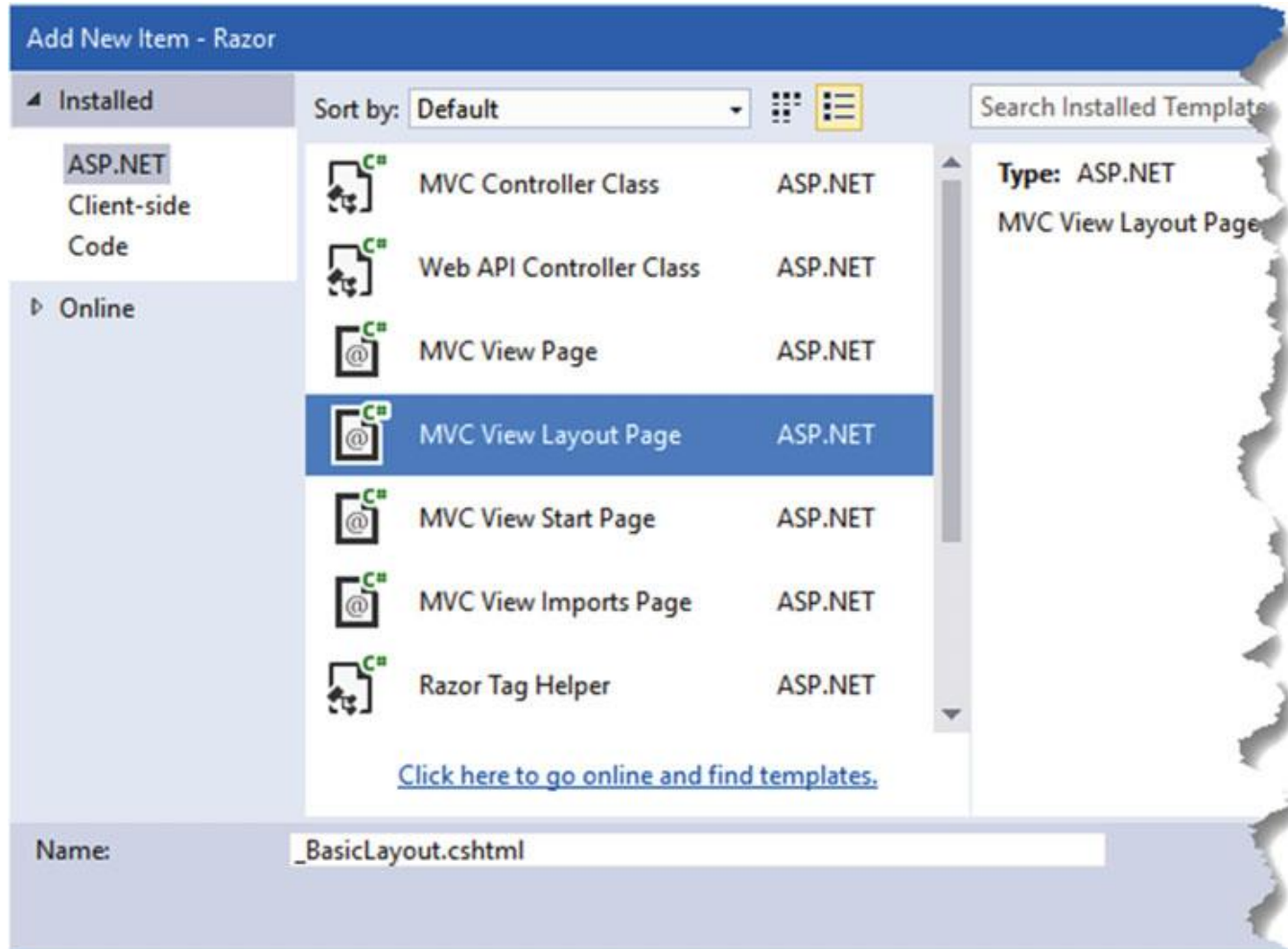
```
@{
```

```
    Layout = "_Layout";
```

```
}
```

WORKING WITH LAYOUTS

- Layouts are typically shared by views used by multiple controllers and are stored in a folder called `Views/Shared`, which is one of the locations that Razor looks in when it tries to find a file.
- Like view import files, the names of layout files begin with an underscore, because they are not meant to be returned directly to the user.

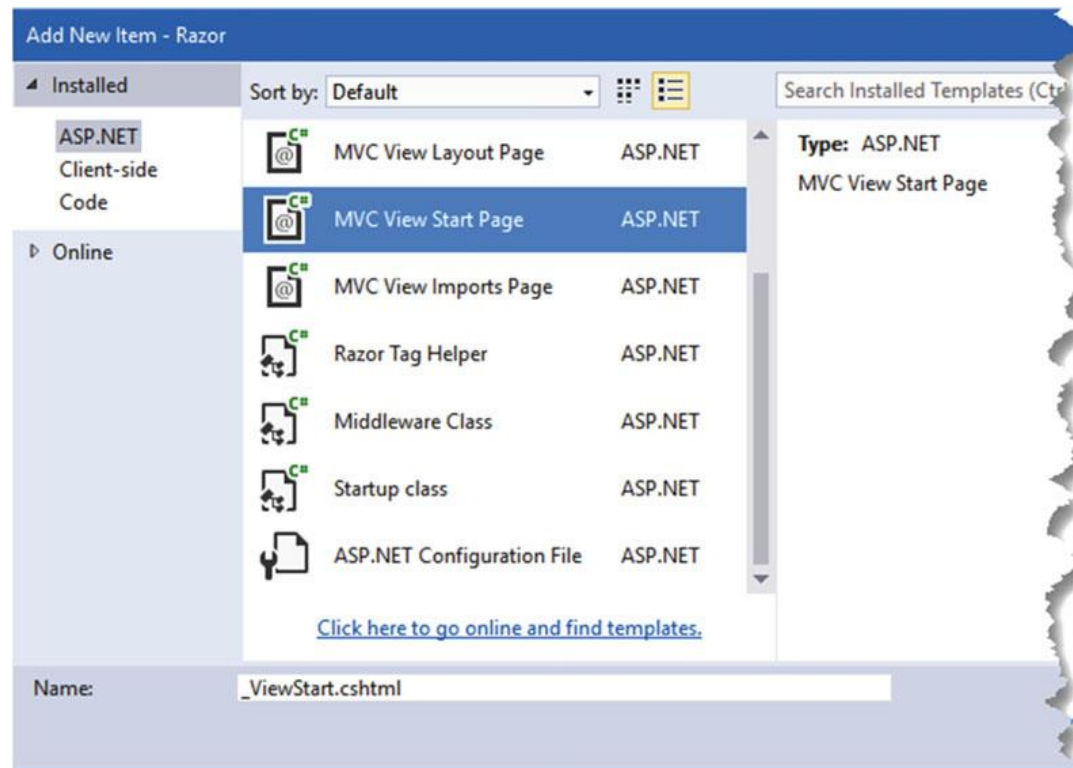


CREATE A VIEW
LAYOUT PAGE

LAYOUTS

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

CREATE A VIEW START PAGE



SPORTS STORE PROJECT



PRODUCT MODEL

```
namespace SportsStore.Models {  
    public class Product {  
        public int ProductID { get; set; }  
        public string Name { get; set; }  
        public string Description { get; set; }  
        public decimal Price { get; set; }  
        public string Category { get; set; }  
    }  
}
```


REPOSITORY INTERFACE

```
namespace SportsStore.Models {  
    public interface IProductRepository {  
        IEnumerable<Product> Products { get; }  
    }  
}
```

FAKE REPOSITORY

```
namespace SportsStore.Models {  
    public class FakeProductRepository : IProductRepository {  
        public IEnumerable<Product> Products => new List<Product> {  
            new Product { Name = "Football", Price = 25 },  
            new Product { Name = "Surf board", Price = 179 },  
            new Product { Name = "Running shoes", Price = 95 }  
        };  
    }  
}
```

SERVICE COMPONENTS

MVC emphasizes the use of loosely coupled components, which means that you can make a change in one part of the application without having to make corresponding changes elsewhere.

This approach categorizes parts of the application as services , which provide features that other parts of the application use.

The class that provides a service can then be altered or replaced without requiring changes in the classes that use it.

CONFIGURING SERVICES

```
public class Startup {  
    // ...  
  
    public void ConfigureServices(IServiceCollection services) {  
        services.AddMvc();  
        services.AddTransient<IProductRepository, FakeProductRepository>();  
    }  
}
```

DEPENDENCY INJECTION

```
public class ProductController : Controller {  
    private IProductRepository repository;  
  
    public ProductController(IProductRepository repository) {  
        this.repository = repository;  
    }  
  
    public ViewResult List() => View(repository.Products);  
}
```

LIST VIEW

```
@model IEnumerable<Product>
```

```
@{  
    ViewData["Title"] = "Products";  
}
```

```
@foreach (var p in Model) {  
    <div>  
        <h3>@p.Name</h3>  
        @p.Description  
        <h4>@p.Price.ToString("c")</h4>  
    </div>  
}
```

ENTITY FRAMEWORK CORE (EF CORE)

[HTTPS://DOCS.MICROSOFT.COM/EN-US/EF/](https://docs.microsoft.com/en-us/ef/)

- EF Core is an object-relational mapping (ORM) framework. An ORM framework presents the tables, columns, and rows of a relational database through regular objects.

CREATING THE DATABASE CLASSES

The database context class is the bridge between the application and the EF Core and provides access to the application's data using model objects.

```
public class ApplicationDbContext : DbContext {  
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) {}  
  
    public DbSet<Product> Products { get; set; }  
}
```


EF PRODUCT REPOSITORY

```
public class EFProductRepository : IProductRepository {  
    private ApplicationDbContext context;  
  
    public EFProductRepository(ApplicationDbContext context) {  
        this.context = context;  
    }  
  
    public IEnumerable<Product> Products => context.Products;  
}
```

CONNECTION STRING (APPSETTINGS.JSON)

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "ConnectionStringSportsStore":
    "Server=(localdb)\\mssqllocaldb;Database=SportsStore;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

CONFIGURING EF DATABASE

```
using Microsoft.EntityFrameworkCore;
```

```
// ...
```

```
public void ConfigureServices(IServiceCollection services) {  
    services.AddMvc();  
    services.AddTransient<IProductRepository, EFProductRepository>();  
  
    services.AddDbContext<ApplicationDbContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("ConnectionStringSportsStore"))  
    );  
}
```

SEED DATA

```
public static class SeedData {  
    public static void EnsurePopulated(IServiceProvider appServices) {  
        ApplicationDbContext context = (ApplicationDbContext) appServices.GetService(typeof(ApplicationDbContext));  
  
        if (context.Products.Any()) return;  
  
        context.Products.AddRange(  
            new Product { Name = "Kayak", Description = "A boat for one person", Category = "Watersports", Price = 275},  
            new Product { Name = "Lifejacket", Description = "Protective and fashionable", Category = "Watersports", Price = 48.95m},  
            new Product { Name = "Soccer Ball", Description = "FIFA-approved size and weight", Category = "Soccer", Price = 19.50m},  
            new Product { Name = "Corner Flags", Description = "Give your playing field a professional touch", Category = "Soccer", Price = 34.95m},  
            new Product { Name = "Stadium", Description = "Flat-packed 35,000-seat stadium", Category = "Soccer", Price = 79500},  
            new Product { Name = "Thinking Cap", Description = "Improve brain efficiency by 75%", Category = "Chess", Price = 16},  
            new Product { Name = "Unsteady Chair", Description = "Secretly give your opponent a disadvantage", Category = "Chess", Price = 29.95m},  
            new Product { Name = "Human Chess Board", Description = "A fun game for the family", Category = "Chess", Price = 75},  
            new Product { Name = "Bling-Bling King", Description = "Gold-plated, diamond-studded King", Category = "Chess", Price = 1200}  
        );  
        context.SaveChanges();  
    }  
}
```

SEED DATA

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    // ...

    SeedData.EnsurePopulated(app.ApplicationServices);
}
```

SEED DATA

- ASP.NET MVC CORE 2

```
namespace SportsStore {  
    public class Program {  
        public static void Main(string[] args) {  
            BuildWebHost(args).Run();  
        }  
  
        public static IWebHost BuildWebHost(string[] args) =>  
            WebHost.CreateDefaultBuilder(args)  
                .UseStartup<Startup>()  
                .UseDefaultServiceProvider(options =>  
                    options.ValidateScopes = false)  
                .Build();  
    }  
}
```

DATABASE MIGRATION

Entity Framework Core is able to generate the schema for the database using the model classes through a feature called migrations.

When you prepare a migration, EF Core creates a C# class that contains the SQL commands required to prepare the database. If you need to modify your model classes, then you can create a new migration that contains the SQL commands required to reflect the changes.

In this way, you don't have to worry about manually writing and testing SQL commands and can just focus on the C# model classes in the application.

EF Core commands are performed using the Package Manager Console.

PREPARE THE DATABASE FOR ITS FIRST USE

- Add-Migration Initial
- Update-Database

ROUTING

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

PAGINATION SUPPORT

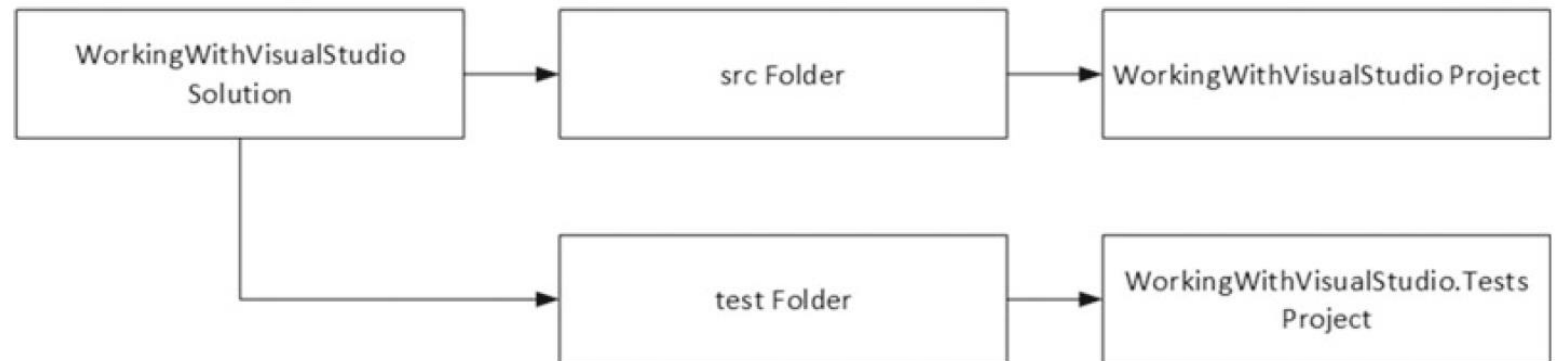
```
public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;

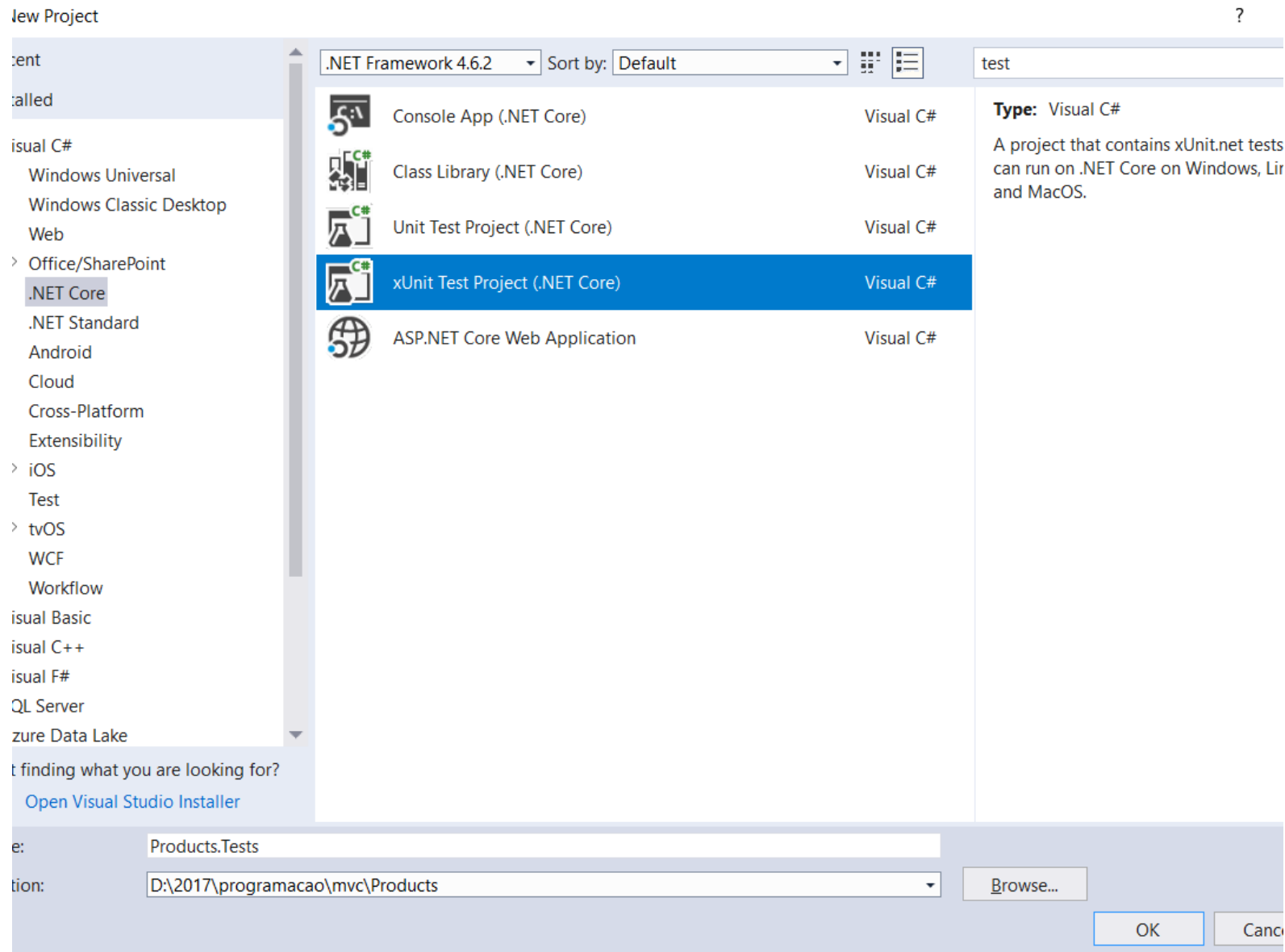
    public ProductController(IProductRepository repository) {
        this.repository = repository;
    }

    public ViewResult List(int page = 1) => View(
        repository.Products
            .OrderBy(p => p.Price)
            .Skip((page - 1) * PageSize)
            .Take(PageSize)
    );
}
```

UNIT TESTING

- Unit tests are used to validate the behavior of individual components and features in an application, and ASP.NET Core and ASP.NET Core MVC have been designed to make it as easy as possible to set up and run unit tests for web applications.
- For ASP.NET Core applications, you generally create a separate Visual Studio project to hold the unit tests, each of which is defined as a method in a C# class. Using a separate project means you can deploy your application without also deploying the tests.





UNIT TESTS

Projects

Solution

Shared Projects

Browse

Search (Ctrl+E)

	Name	Path
<input type="checkbox"/>	Products	D:\2017\programacao...



Name:
Products

Browse...

OK

Cancel

ADD REFERENCE



```
public class ProductControllerTests {  
    [Fact]  
    public void CanPaginate() {  
        // Arrange  
        // Act  
        // Assert  
    }  
}
```

MOCK

- Moq creates fake implementations of components in an application.
- It makes it easier to create fake components to isolate parts of the application for unit testing.
- <https://www.nuget.org/packages/moq>



```
[Fact]
public void CanPaginate() {
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    });

    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Act
    // Assert
}
```




```
using System.Linq;
```

```
[Fact]
```

```
public void CanPaginate() {
```

```
    // Arrange
```

```
    // ...
```

```
    // Act
```

```
    IEnumerable<Product> result = controller.List(2).ViewData.Model as IEnumerable<Product>;
```

```
    // Assert
```

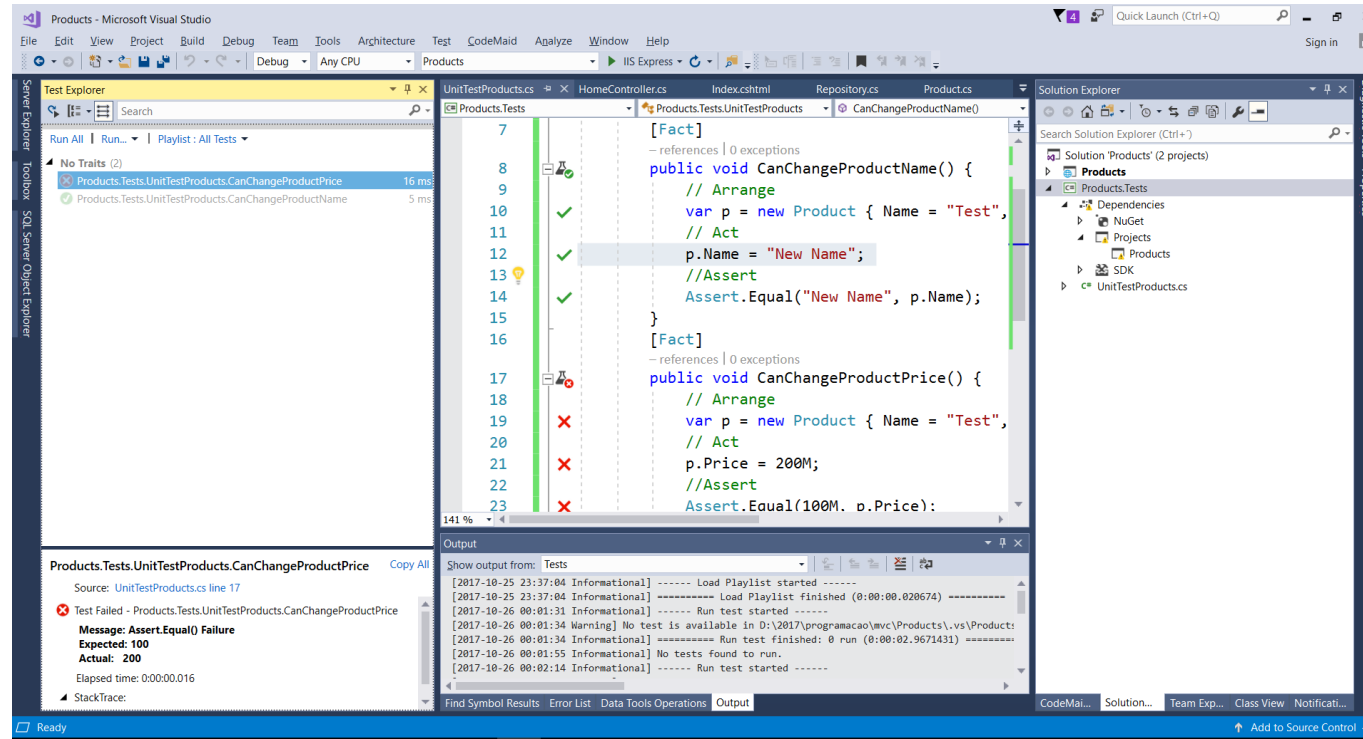
```
    Product[] prodArray = result.ToArray();
```

```
    Assert.True(prodArray.Length == 2);
```

```
    Assert.Equal("P4", prodArray[0].Name);
```

```
    Assert.Equal("P5", prodArray[1].Name);
```

```
}
```



RUNNING TESTS

UNIT TESTS

- When a test fails, it is always a good idea to check the accuracy of the test before looking at the component it targets, especially if the test is new or has been recently modified.

PAGINGINFO VIEWMODEL

```
namespace SportsStore.Models.ViewModels {  
    public class PagingInfo {  
        public int TotalItems { get; set; }  
        public int ItemsPerPage { get; set; }  
        public int CurrentPage { get; set; }  
        public int TotalPages =>  
            (int) Math.Ceiling((decimal) TotalItems / ItemsPerPage);  
    }  
}
```

TAG HELPERS

```
namespace SportsStore.Infrastructure {
    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PagingLinksTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PagingLinksTagHelper(IUrlHelperFactory helperFactory) { urlHelperFactory = helperFactory; }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        // ...
    }
}
```

TAG HELPERS

```
public class PageLinkTagHelper : TagHelper {  
    // ...  
  
    public override void Process(TagHelperContext context, TagHelperOutput output) {  
        IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);  
  
        TagBuilder result = new TagBuilder("div");  
        for (int i = 1; i <= PageModel.TotalPages; i++) {  
            TagBuilder tag = new TagBuilder("a");  
            tag.Attributes["href"] = urlHelper.Action(PageAction, new { page = i });  
            tag.InnerHtml.Append(i.ToString());  
  
            result.InnerHtml.AppendHtml(tag);  
        }  
        output.Content.AppendHtml(result.InnerHtml);  
    }  
}
```

PRODUCTS LIST VIEW MODEL

```
namespace SportsStore.Models.ViewModels {  
    public class ProductsListViewModel {  
        public IEnumerable<Product> Products { get; set; }  
        public PagingInfo PagingInfo { get; set; }  
    }  
}
```

PRODUCT CONTROLLER

```
public ActionResult List(int page = 1) =>
    View(
        new ProductsListViewModel {
            Products = repository.Products
                .OrderBy(p => p.Price)
                .Skip((page - 1) * PageSize)
                .Take(PageSize),

            PagingInfo = new PagingInfo {
                CurrentPage = page,
                ItemsPerPage = PageSize,
                TotalItems = repository.Products.Count()
            }
        }
    );
```


ADDING THE INFRASTRUCTURE TAG HELPERS

```
@using SportsStore
```

```
@using SportsStore.Models
```

```
@using SportsStore.Models.ViewModels
```

```
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

VIEW

```
@model ProductsListViewModel

@{
    ViewData["Title"] = "Products";
}

@foreach (var p in Model.Products) {
    <div class="well well-sm">
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="List"></div>
```

BOOKS PROJECT



AUTHOR CLASS

```
namespace Books.Models {  
    public class Author {  
        public int AuthorId { get; set; }  
        public string Name { get; set; }  
        public ICollection<Book> Books { get; set; }  
    }  
}
```

BOOK MODEL

```
namespace Books.Models {  
    public class Book {  
        public int BookId { get; set; }  
        public string Title { get; set; }  
        public Author Author { get; set; }  
        public int AuthorId { get; set; }  
        public ICollection<BookCategory> Categories { get; set; }  
    }  
}
```

CATEGORY MODEL

```
namespace Books.Models {  
    public class Category {  
        public int CategoryId { get; set; }  
        public string Name { get; set; }  
        public ICollection<BookCategory> Books { get; set; }  
    }  
}
```

BOOKCATEGORY MODEL

```
namespace Books.Models {  
    public class BookCategory {  
        public int BookId { get; set; }  
        public Book Book { get; set; }  
        public int CategoryId { get; set; }  
        public Category Category { get; set; }  
    }  
}
```

DATABASE CONTEXT

```
public class BooksDbContext : DbContext {  
    public BooksDbContext(  
        DbContextOptions<BooksDbContext> options) : base(options) {  
    }  
  
    public DbSet<Author> Authors;  
    public DbSet<Book> Books;  
    public DbSet<Category> Categories;  
    public DbSet<BookCategory> BooksCategories;  
}
```


DATABASE CONTEXT: FLUENT API FOR EF CORE

```
public class BooksDbContext : DbContext {  
    // ...  
  
    protected override void OnModelCreating(ModelBuilder modelBuilder) {  
        modelBuilder.Entity<BookCategory>()  
            .HasKey(bc => new { bc.BookId, bc.CategoryId });  
  
        modelBuilder.Entity<BookCategory>()  
            .HasOne(bc => bc.Book)  
            .WithMany(b => b.BookCategories)  
            .HasForeignKey(bc => bc.BookId);  
  
        modelBuilder.Entity<BookCategory>()  
            .HasOne(bc => bc.Category)  
            .WithMany(c => c.BookCategories)  
            .HasForeignKey(bc => bc.CategoryId);  
    }  
}
```

CONNECTION STRING (APPSETTINGS.JSON)

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "ConnectionStringBooks":
      "Server=(localdb)\\mssqllocaldb;Database=Books;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

CONFIGURING EF DATABASE

```
using Microsoft.EntityFrameworkCore;
```

```
// ...
```

```
public void ConfigureServices(IServiceCollection services) {  
    services.AddMvc();  
    services.AddDbContext<BooksDbContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("ConnectionStringBooks"))  
    );  
}
```

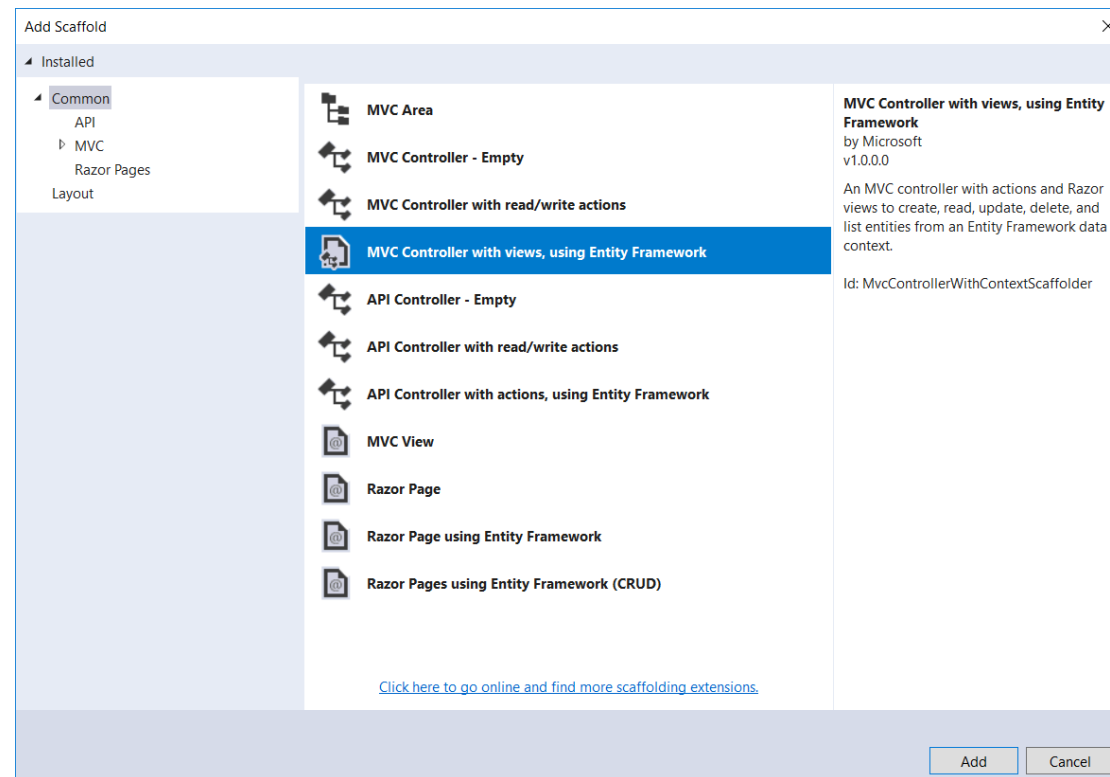
PREPARE THE DATABASE FOR ITS FIRST USE

- Add-Migration Initial
- Update-Database

SCAFFOLDING

You can sit down and think about the problem you want to solve, and write plain C# classes, such as Album, ShoppingCart, and User, to represent the primary objects involved. When you are ready, you can then use tools provided by MVC to construct the controllers and views for the standard index, create, edit, and delete scenarios for each of the model objects. The construction work is called *scaffolding*.

SCAFFOLDING



SCAFFOLDING

Add MVC Controller with views, using Entity Framework

Model class:

Book (Books.Models)

Data context class:

BooksDbContext (Books.Data)

+

Views:

☒ Generate views

☒ Reference script libraries

☒ Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

Controller name:

BooksController

Add

Cancel

SPORTS STORE PROJECT



STYLING THE PRODUCTS LIST

```
@foreach (var p in Model.Products) {  
    <div class="well well-sm">  
        <h3>  
            <strong>@p.Name</strong>  
            <span class="pull-right label label-primary">@p.Price.ToString("c")</span>  
        </h3>  
        @p.Description  
  
    </div>  
}  
  
<div page-model="@Model.PagingInfo" page-action="List"></div>
```

```
public class PageLinkTagHelper : TagHelper {  
    // ...  
  
    public bool CssClassesEnabled { get; set; } = false;  
    public string CssClassPage { get; set; }  
    public string CssClassPageNormal { get; set; }  
    public string CssClassPageSelected { get; set; }  
}
```

TAG HELPERS

```
public override void Process(TagHelperContext context, TagHelperOutput output) {
    IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);

    TagBuilder result = new TagBuilder("div");
    for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        tag.Attributes["href"] = urlHelper.Action(PageAction, new { page = i });
        tag.InnerHtml.Append(i.ToString());

        if (PageClassesEnabled) {
            tag.AddCssClass(PageClass);
            tag.AddCssClass(i == PageModel.CurrentPage ? PageClassSelected : PageClassNormal);
        }

        result.InnerHtml.AppendHtml(tag);
    }
    output.Content.AppendHtml(result.InnerHtml);
}
```

STYLING THE PRODUCTS LIST

```
@foreach (var p in Model.Products) {  
    <div class="well well-sm">  
        <h3>  
            <strong>@p.Name</strong>  
            <span class="pull-right label label-primary">@p.Price.ToString("c")</span>  
        </h3>  
        @p.Description  
    </div>  
}  
  
<div page-model="@Model.PagingInfo" page-action="List"  
    page-classes-enabled="true" page-class="btn" page-class-normal="btn-default"  
    page-class-selected="btn-primary" class="btn-group pull-right">  
</div>
```

PARTIAL VIEWS

@model Product

```
<div class="well well-sm">
  <h3>
    <strong>@Model.Name</strong>
    <span class="pull-right label label-primary">@Model.Price.ToString("c")</span>
  </h3>
  @Model.Description
</div>
```

USING PARTIAL VIEWS

```
@model ProductsListViewModel
```

```
@{  
    ViewData["Title"] = "Products";  
}
```

```
@foreach (var p in Model.Products) {  
    @Html.Partial("ProductSummary", p)  
}
```