ASP.NET Core

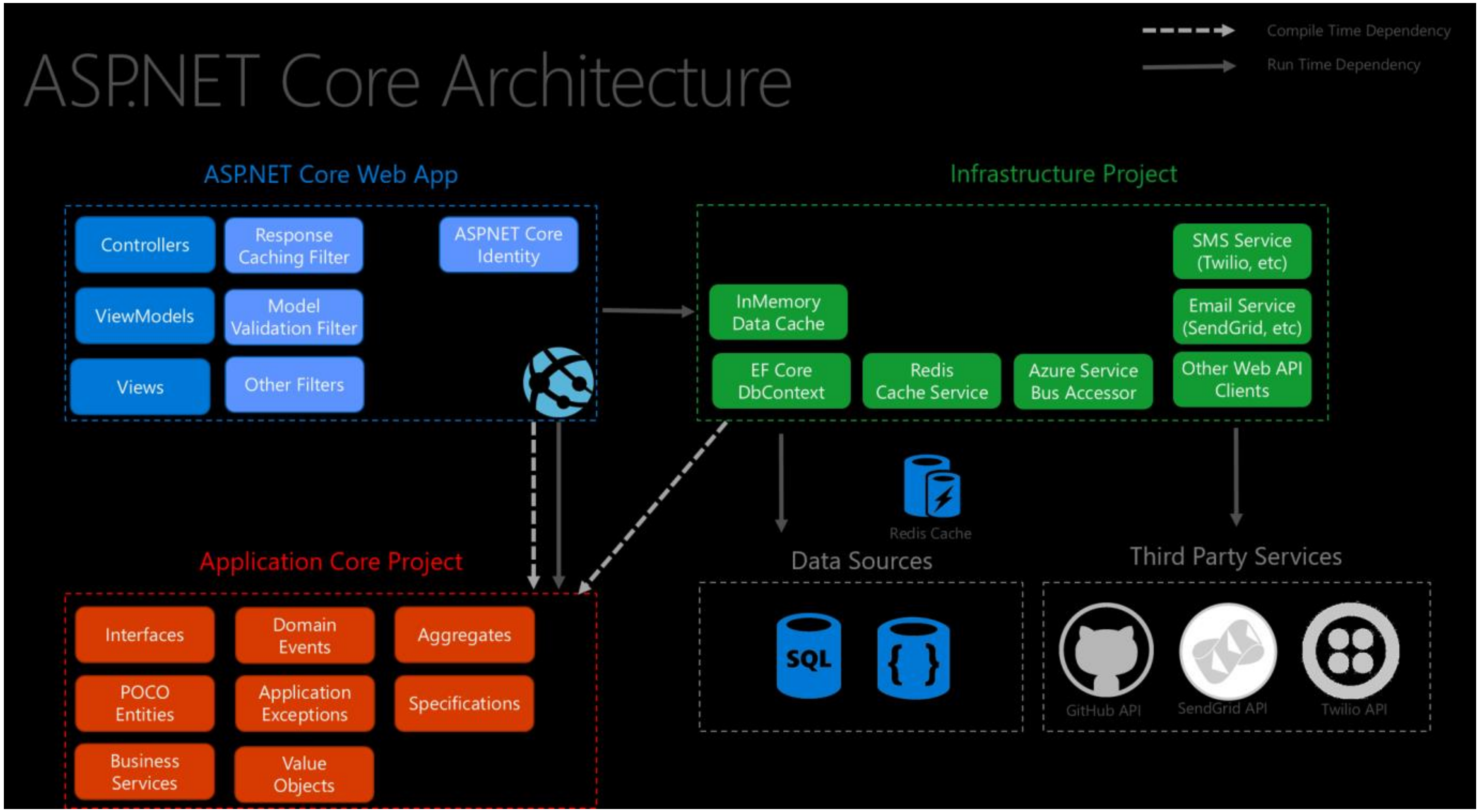
ASP.NET Core

- ASP.NET MVC: Microsoft's implementation of the MVC architectural pattern, used for building test-friendly web applications that are robust yet lightweight.
- Web API: Microsoft's answer to RESTful APIs, Web API allows developers to build HTTP services that can serve as a backend to web applications and mobile apps.
- Web Pages: Microsoft's solution for creating lightweight dynamic websites. At the time of writing, Web Pages is not available in ASP .NET Core 1.0 so it will not be covered in this book.

The .NET Core

- The new .NET Core is a lightweight cross-platform subset of the full .NET Framework that makes its home on Windows, Linux, and OS X.

ASP.NET Core Architecture



* You can find different web frameworks architectures that have similar functionality.

https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures

ASP.NET Core MVC includes the following:

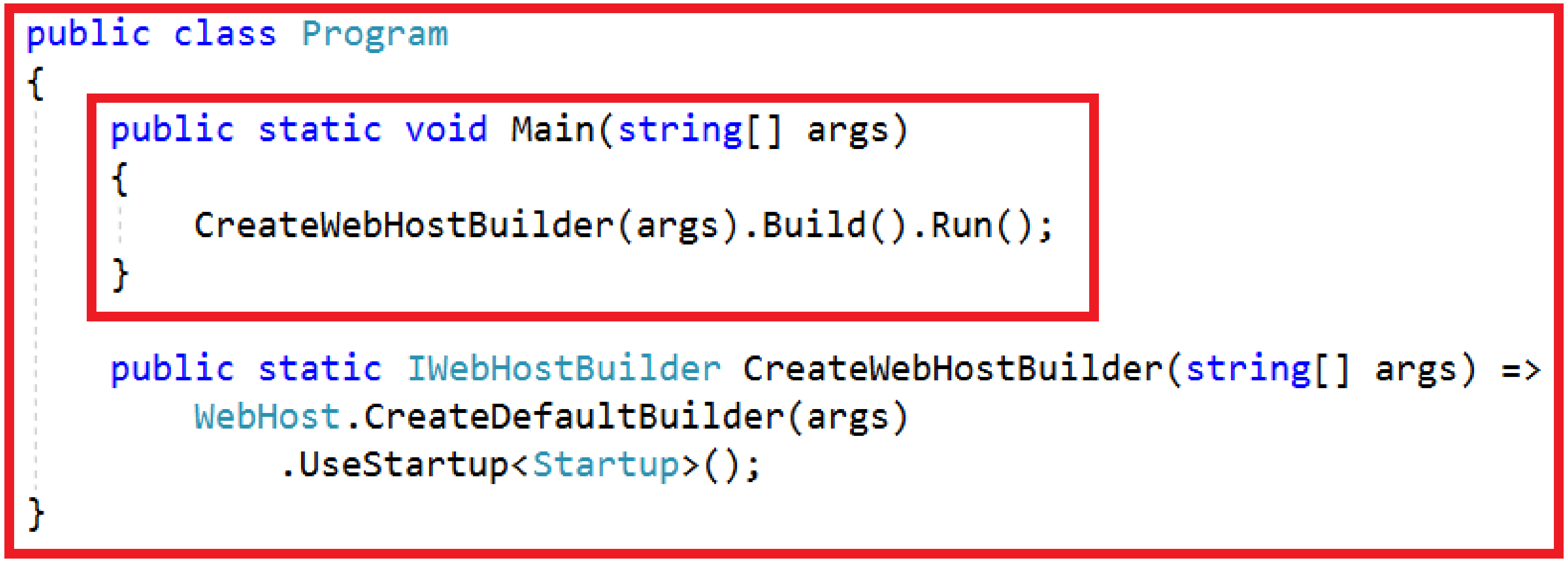
- Model – View - Controller
- Routing
- Model binding
- Model validation
- Dependency injection
- Filters
- Areas
- Web APIs
- Testability
- Razor view engine
- Strongly typed views
- Tag Helpers
- View Components

Entry Point of ASP web appicaton

- The most important point that you need to keep in mind is, the ASP.NET Core Application initially starts as a Console Application and the Main() method is the entry point to the application.

- So, when we execute the ASP.NET Core application, first it looks for the Main() method and this is the method from where the execution starts. The Main() method then configures the ASP.NET Core and starts it. At this point, the application becomes an ASP.NET Core web application.

Program.cs



```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

https://dotnettutorials.net/lesson/asp-net-core-main-method/

Startup.cs

- While setting up the web host, the Startup class is also configured using the UseStartup() extension method of IWebHostBuilder class.

Methods

- The Configure() method can be used to configure the HTTP request pipeline by enabling static files, Identity, MVC, and others.
- It can be used to set a default route inside the Startup.cs class, replacing the need to have a global.asax file.
- Note that the Configure() method is called after the ConfigureServices() method, which can be used to add various services, such as Entity Framework, Identity, and MVC. Both of these methods are called by the runtime.

InProcess Hosting

- When we create a new ASP.NET Core Web application by using any template, by default the project file is created with InProcess hosting which is used for hosting the application in IIS or IIS Express scenarios.

- In case of InProcess hosting (i.e. when the CreateDefaultBuilder() sees the value as InProcess for the AspNetCoreHostingModel element in the project file), behind the scene the CreateDefaultBuilder() method internally calls the UseIIS() method. Then host the application inside the IIS worker process (i.e. w3wp.exe for IIS and iisexpress.exe for IISExpress).

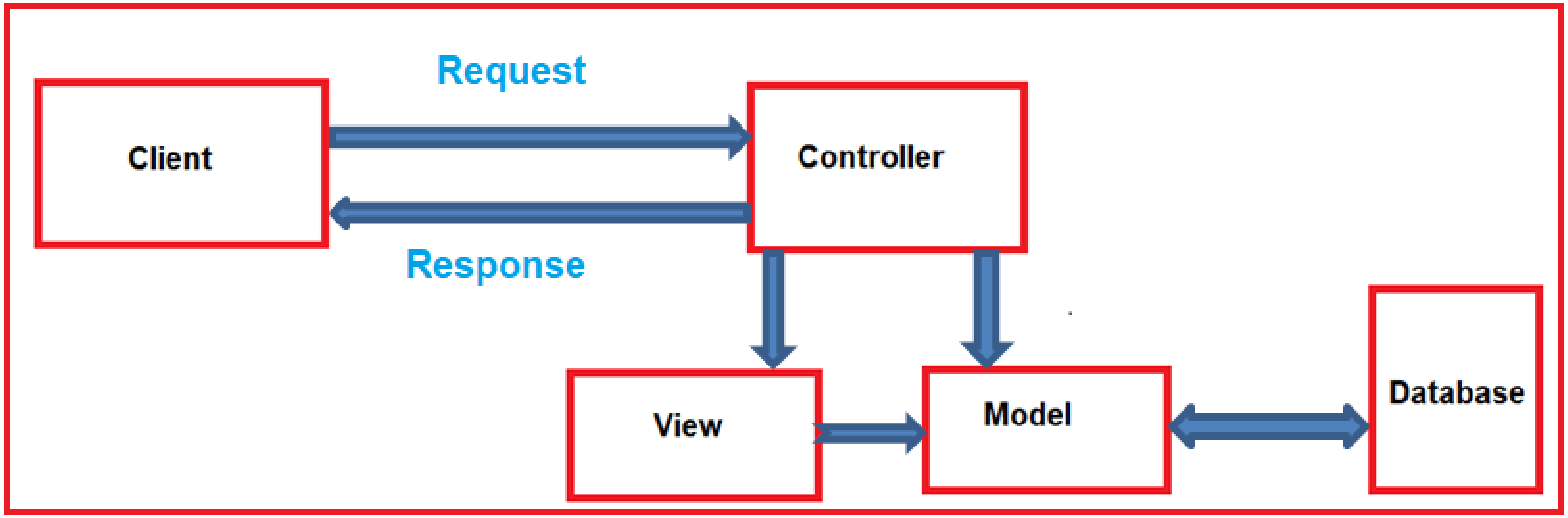
- From the performance point of view, the InProcess hosting model delivers significantly higher request throughput than the OutOfProcess hosting model. Why we will discuss at the end of this article.

Traditional Application vs Single Page Application (SPA)

Factor	Traditional Web App	Single Page Application
Required Team Familiarity with JavaScript/TypeScript	Minimal	Required
Support Browsers without Scripting	Supported	Not Supported
Minimal Client-Side Application Behavior	Well-Suited	Overkill
Rich, Complex User Interface Requirements	Limited	Well-Suited

https://www.c-sharpcorner.com/article/asp-net-core-2-architecture-design-pattern-ideology/

ASP.Core MVC



https://dotnettutorials.net/lesson/introduction-asp-net-core-mvc/

MVC design pattern

- The main objective of the MVC design pattern is the separation of concerns. It means the **domain mode**l and **business logic** are separated from the **user interface (i.e. view)**.

Controller

- Each controller is responsible for handling user requests, based on a matching route. Controllers can update data in a model, and then select a view to return back to the user.
- In MVC, each controller typically returns an IActionResult from its action methods. For web application projects, this could be a view. For Web API projects, this could be a set of data. It is possible to return both views and results from a controller.

Data flow from the Controller to the View

- Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests.
- The controller retrieves data from a data source and decides what type of response to send back to the browser.
- View templates can be used from a controller to generate and format an HTML response to the browser.

Controller: HTTP Methods

- Whenever a client submits a request to a server, part of that request is an HTTP method, which is what the client would like the server to do with the specified resource.

The GET Method

- GET is used to request data from a specified resource.
- GET is one of the most common HTTP methods.
- Note that the query string (name/value pairs) is sent in the URL of a GET request:
- /test/demo_form.php?name1=value1&name2=value2

POST vs PUT

- POST and PUT are very similar in that they both send data to the server that the server will need to store somewhere.
- Technically speaking, you could use either for the Create or Update scenarios, and in fact this is rather common.

https://exceptionnotfound.net/using-http-methods-correctly-in-asp-net-web-api/

PUT

- PUT is idempotent. What this means is that if you make the same request twice using PUT, with the same parameters both times, the second request will have no effect. This is why PUT is generally used for the Update scenario; calling Update more than once with the same parameters doesn't do anything more than the first call did.

POST

- By contrast, POST is not idempotent; making the same call using POST with same parameters each time will cause two different things to happen, hence why POST is commonly used for the Create scenario (submitting two identical items to a Create method should create two entries in the data store).

HTTP DELETE

- The HTTP DELETE request is used to delete an existing record in the data source in the RESTful architecture.

Methods and HTTP Request

- If there is an attribute applied (via [HttpGet], [HttpPost], [HttpPut], [AcceptVerbs], etc), the action will accept the specified HTTP method(s).
- If the name of the controller action starts the words "Get", "Post", "Put", "Delete", "Patch", "Options", or "Head", use the corresponding HTTP method.
- Otherwise, the action supports the POST method.

Models

- Model represents domain specific data and business logic in MVC architecture. It maintains the data of the application. Model objects retrieve and store model state in the persistence store like a database.

https://www.tutorialsteacher.com/mvc/mvc-model

Adding a Model

- Similar to adding a class

```
namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set;  }
        public int Age { get; set;  }
    }
}
```

Repository

- You can use a repository pattern with a service layer for models that reflect your database entities through Entity Framework. This will let you have UI elements that don't have to rely on the structure of your database entities.

Reposity Sample

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

Dependency injection in ASP.NET Core

- ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are registered in the app's Startup.ConfigureServices method.
- It is used to inject services such as database to an application

https://docs.microsoft.com/tr-tr/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.210

Dependency Injection of Database Service

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

https://docs.microsoft.com/tr-tr/aspnet/core/tutorials/razor-pages/model?view=aspnetcore-2.2&tabs=visual-studio#pmc

Views

- In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction.

https://docs.microsoft.com/tr-tr/aspnet/core/mvc/views/overview?view=aspnetcore-2.2

Benefits of using views

- Views help to establish separation of concerns within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:
- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a view

- Views that are specific to a controller are created in the Views/[ControllerName] folder.

Creating a view

- CSHTML

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor markup

- Razor markup starts with the @ symbol. Run C# statements by placing C# code within Razor code blocks set off by curly braces ({ ... }).
- For example, see the assignment of "About" to ViewData["Title"] shown above. You can display values within HTML by simply referencing the value with the @ symbol. See the contents of the <h2> and <h3> elements above.
- The view content shown above is only part of the entire webpage that's rendered to the user. The rest of the page's layout and other common aspects of the view are specified in other view files. To learn more, see the Layout topic.

What are Tag Helpers

- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files

Tag Helper Example

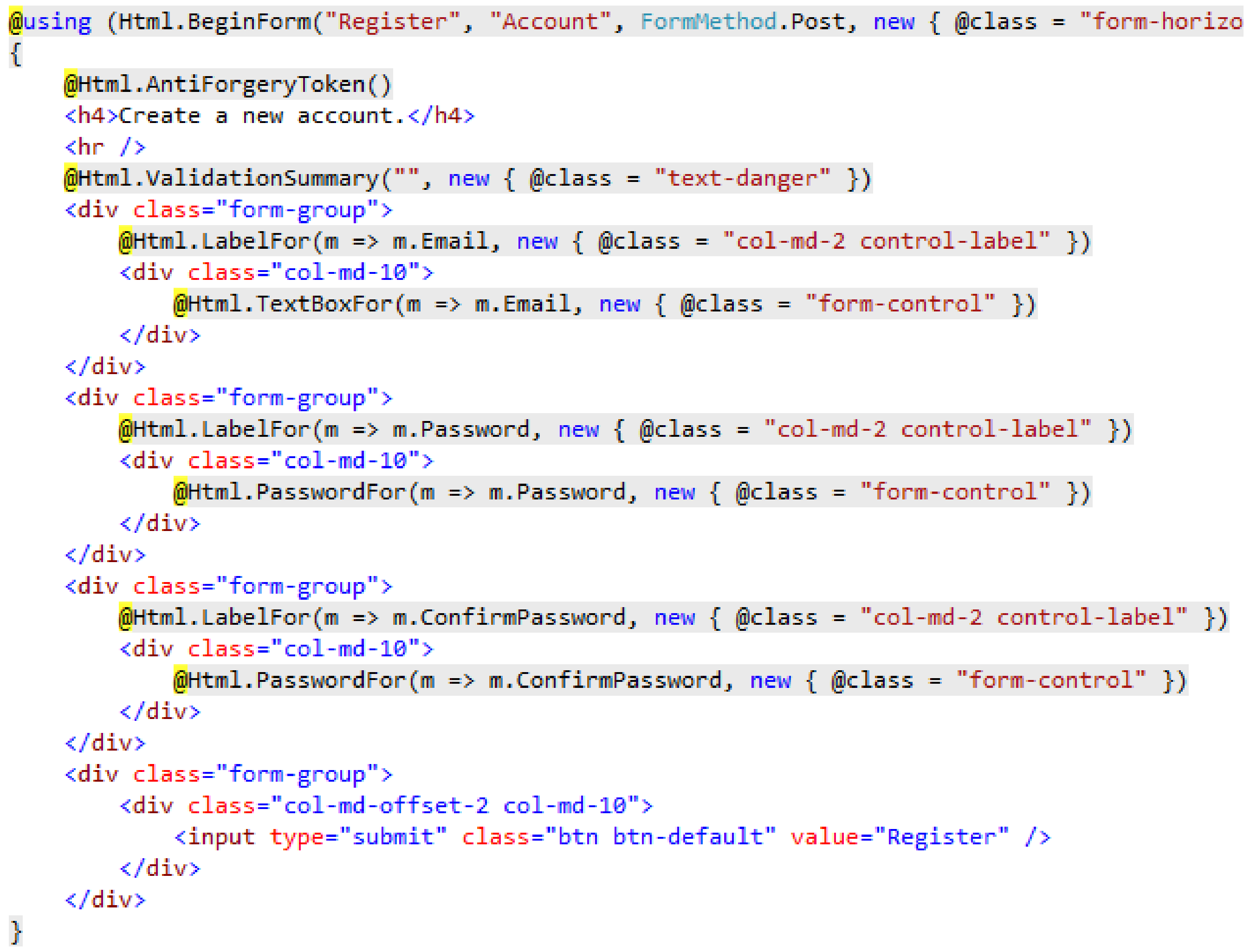
C#
```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
}
```

- The following Razor markup:

CSHTML
```
<label asp-for="Movie.Title"></label>
```
Generates the following HTML:

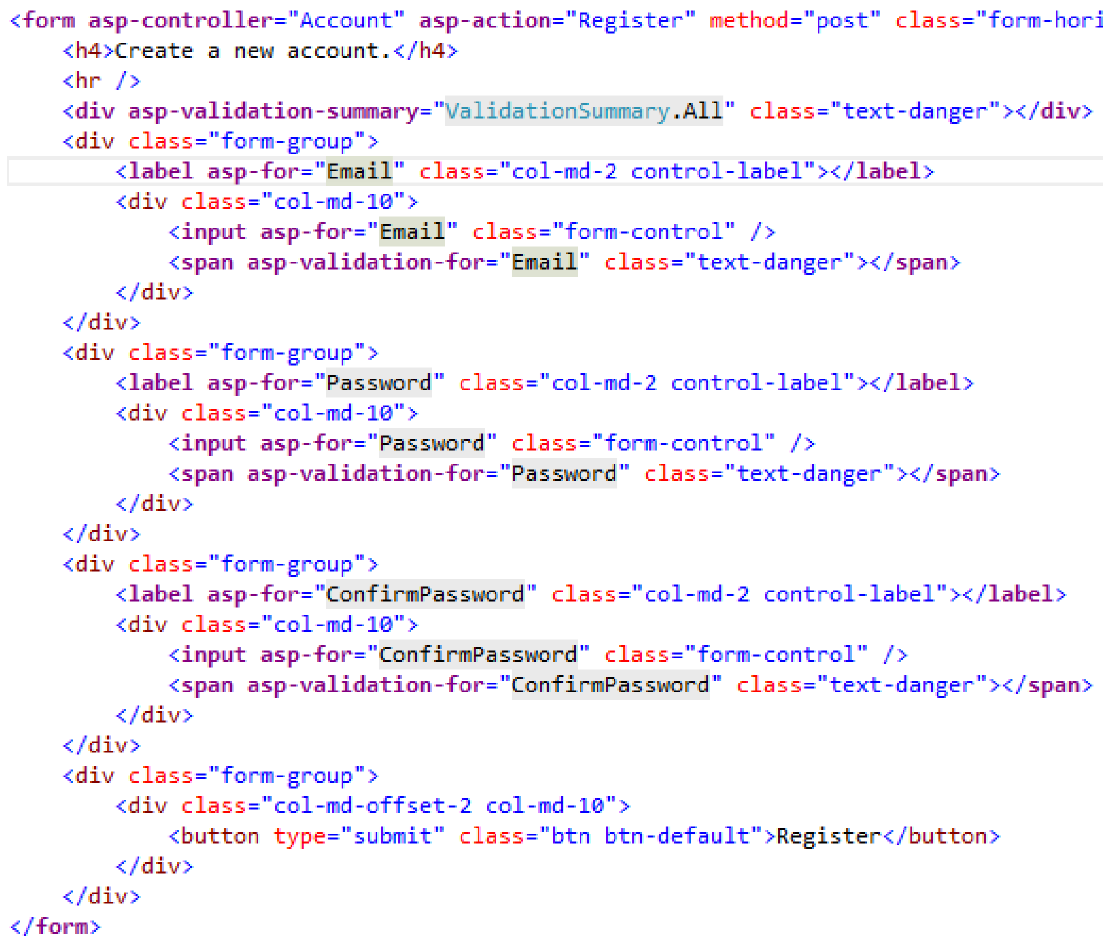
HTML
```
<label for="Movie_Title">Title</label>
```

Razor Example



```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

Equivalent approach using Tag Helpers



```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

The Form Tag Helper

- Generates the HTML <FORM> action attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the [ValidateAntiForgeryToken] attribute in the HTTP Post action method)
- Provides the asp-route-<Parameter Name> attribute, where <Parameter Name> is added to the route values. The routeValues parameters to Html.BeginForm and Html.BeginRouteForm provide similar functionality.
- Has an HTML Helper alternative Html.BeginForm and Html.BeginRouteForm

The Form Tag Helper Example

HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

- The Form Tag Helper above generates the following HTML:

HTML

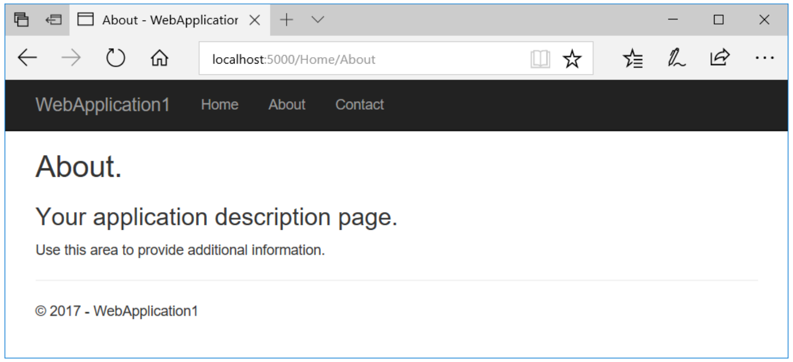
```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

How controllers specify views

- Views are typically returned from actions as a ViewResult, which is a type of ActionResult. Your action method can create and return a ViewResult directly, but that isn't commonly done. Since most controllers inherit from Controller, you simply use the View helper method to return the ViewResult:
- HomeController.cs (C#)
-

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```



About.

Your application description page.

Use this area to provide additional information.

© 2017 - WebApplication1

Passing data to views

- Pass data to views using several approaches:
  - Strongly typed data: viewmodel
  - Weakly typed data
    - ViewData (ViewDataAttribute)
    - ViewBag

Strongly typed data (viewmodel)

- The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.
- Specify a model using the @model directive
- This strongly typed approach enables better compile time checking of your code.

Specify a model using the @model directive

CSHTML

```
@model WebApplication1.ViewModels.Address


<h2>Contact</h2>

<address>

    @Model.Street<br>

    @Model.City, @Model.State
@Model.PostalCode<br>

    <abbr title="Phone">P:</abbr> 425.555.0100

</address>
```

To provide the model to the view, the controller passes it as a parameter:

C#

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

Model Validate

- The model binding system takes the posted form values and creates a Movie object that's passed as the movie parameter.
- The ModelState.IsValid method verifies that the data submitted in the form can be used to modify (edit or update) a Movie object.
- If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the SaveChangesAsync method of database context. After saving the data, the code redirects the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes just made.

Viewmodel

- Usually, viewmodel classes are either stored in the Models folder or a separate ViewModels folder at the root of the app. The Address viewmodel used in the example above is a POCO viewmodel stored in a file named Address.cs:

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

Dynamic views

- Views that don't declare a model type using @model but that have a model instance passed to them (for example, return View(Address);) can reference the instance's properties dynamically:

- CSHTML

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

- This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

Weakly typed data (ViewData, ViewData attribute, and ViewBag)

- In addition to strongly typed views, views have access to a weakly typed (also called loosely typed) collection of data. Unlike strong types, weak types (or loose types) means that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.
- ViewBag isn't available in Razor Pages.

ViewData

- ViewData is a ViewDataDictionary object accessed through string keys.
- String data can be stored and used directly without the need for a cast, but you must cast other ViewData object values to specific types when you extract them.
- You can use ViewData to pass data from controllers to views and within views, including partial views and layouts.

ViewData Example

C#	CSHTML

C#

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

CSHTML

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State
@address.PostalCode
</address>
```

ViewData Object Passing

```
Student student = new Student()
          {
             StudentId = 101,
             Name = "James",
             Branch = "CSE",
             Section = "A",
             Gender = "Male"
          };
          //storing Student Data
          ViewData["Student"] = student;

@{
var student = ViewData["Student"]
as FirstCoreMVCApplication.Models.Student;
}
```

ViewData attribute

- Another approach that uses the ViewDataDictionary is ViewDataAttribute. Properties on controllers or Razor Page models decorated with [ViewData] have their values stored and loaded from the dictionary.

ViewData attribute Example

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Hello World";

        return View();
    }
}
```

CSHTML

```
<h1>@Model.Title</h1>
```

In the layout, the title is read from the ViewData dictionary:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

ViewBag

- ViewBag is a DynamicViewData object that provides dynamic access to the objects stored in ViewData.

```
[Dynamic]
public dynamic ViewBag{get;}
```

- ViewBag can be more convenient to work with, since it doesn't require casting.
- Derives from DynamicViewData, so it allows the creation of dynamic properties using dot notation
- ViewBag isn't available in Razor Pages.

ViewBag Example

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

CSHTML

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State
@ViewBag.Address.PostalCode
</address>
```

ViewBag

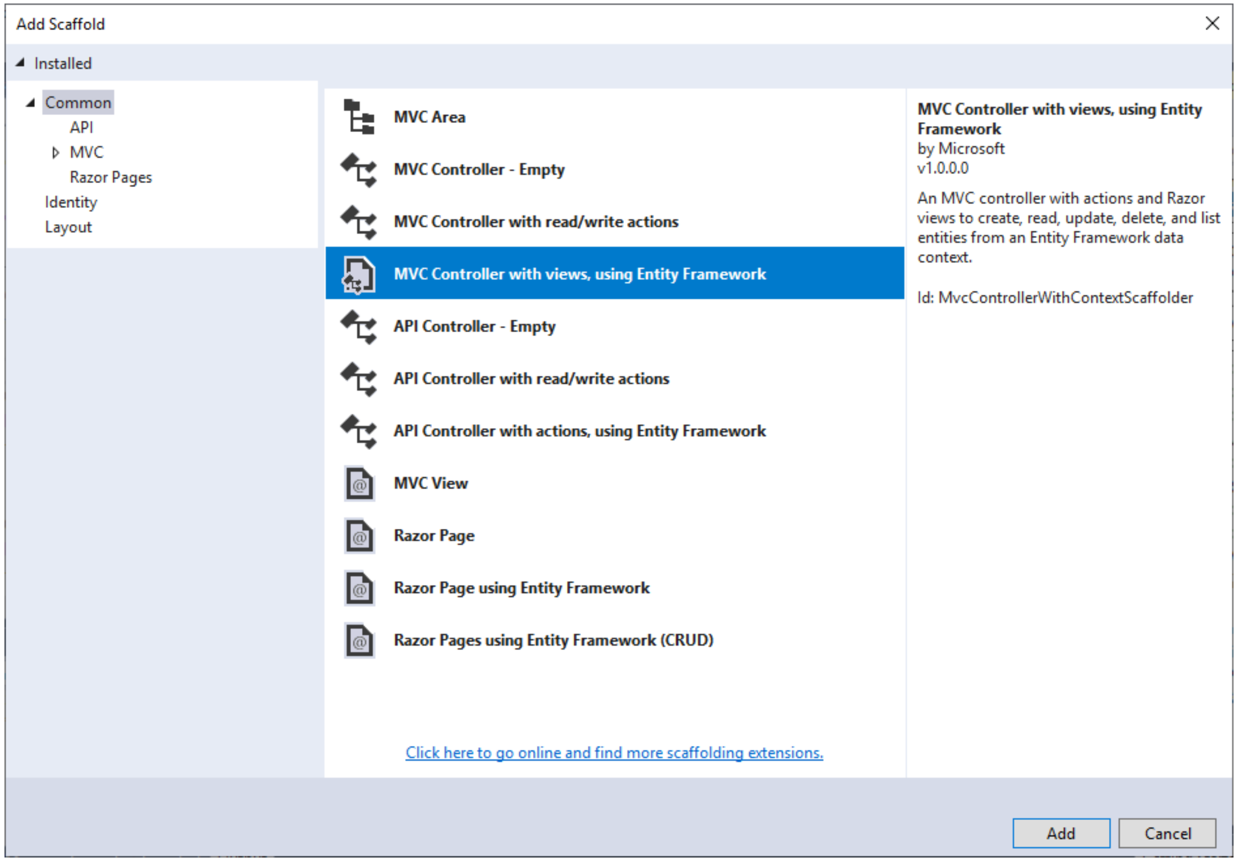
- Typecasting is not required while accessing the data from a ViewBag.
- It does not matter whether the data that we are accessing is of type string or any complex type.

HTML Helper Tags

```
<div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
            </ul>
 </div>
```

Create view from model

- Project → right click→ new or Project→ right click→ Scaffold item
- MVC controller with views using Entity Framework


Add Scaffold
Installed
Common
API
MVC
Razor Pages
Identity
Layout
MVC Area
MVC Controller - Empty
MVC Controller with read/write actions
MVC Controller with views, using Entity Framework
API Controller - Empty
API Controller with read/write actions
API Controller with actions, using Entity Framework
MVC View
Razor Page
Razor Page using Entity Framework
Razor Pages using Entity Framework (CRUD)
Click here to go online and find more scaffolding extensions.
Add
Cancel

MVC Controller with views, using Entity Framework
by Microsoft
v1.0.0.0

An MVC controller with actions and Razor views to create, read, update, delete, and list entities from an Entity Framework data context.

Id: MvcControllerWithContextScaffolder

Routes

- Adding routes to handle anticipated URL paths Back in the days of classic ASP, application URL paths typically reflected physical file paths.
- This continued with ASP.NET web forms, even though the concept of custom URL routing was introduced. With ASP.NET MVC, routes were designed to cater to functionality rather than physical paths.
- ASP.NET Web API has the ability to set up custom routes from within your code. You can create routes for your application using fluent configuration in your startup code or with declarative attributes surrounded by square brackets.

Routes

- To understand the purpose of having routes , let's focus on the features and benefits of routes in your application.
- Using Web APIs to Extend Your Application Human-readable route paths are also SEO-friendly, which is great for Search Engine Optimization
- It provides some level of obscurity when it comes to revealing the underlying web technology and physical file names in your system

Create first Application

- https://docs.microsoft.com/tr-tr/visualstudio/ide/quickstart-aspnet-core?view=vs-2017

Model Binding in ASP.NET Core

- Controllers and Razor pages work with data that comes from HTTP requests.
- For example, route data may provide a record key, and posted form fields may provide values for the properties of the model.
- Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone.
- Model binding automates this process.
- Model Binding extracts the data from an HTTP request and provides them to the controller action method parameters.

https://docs.microsoft.com/tr-tr/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.0

HTTP Request Data Sources

- **Form values:** Values in the FORM in HTTP POST requests.
- **Route values:** Values provided by the Routing system.
- **Query string:** Values found in the URL's query string (e.g. after ? character).

Type conversion errors

- If a source is found but can't be converted into the target type, model state is flagged as invalid.
- In a Razor page, redisplay the page with an error message:

```
if (!ModelState.IsValid) {
    return Page();
}
```

Useful Examples

- https://docs.microsoft.com/tr-tr/ef/core/get-started/aspnetcore/existing-db
- https://www.youtube.com/watch?v=z3YOd0ZNT9o

Questions