# ASP.NET CORE

## WHAT IS ASP.NET CORE?

* ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud based, Internet-connected applications
* ASP.NET Core is a redesign of ASP.NET 4.x

## ASP.NET CORE BENEFITS AND FEATURES



### CROSS PLATFORM:

1. ASP.NET Core applications can be developed and run across different platforms like Windows, MacOS, Linux
2. ASP.NET Core applications can be hosted on IIS, Apache, Docker, Self-host in your own process

### One Unified Programming Model for MVC and Web API:

Both the MVC Controller class and the ASP.NET Web API Controller class inherit from the same 'Controller' base class and returns 'IActionResult'



### Modular:

* ASP.NET Core Provides Modularity with Middleware Components
* Both the request and response pipelines are composed using the middleware components
* Rich set of built-in middleware components are provided out of the box
* Custom Middleware Components can also be created

## ASP.NET Core Project File

### TargetFramework

* Specifies the target framework for the application
* To specify a target framework we use Target Framework Moniker (TFM)



In our case the TFM is **netcoreapp3.1**

### AspNetCoreHostingModel

* Specifies how the application should be hosted
* InProcess or OutOfProcess
* InProcess hosts the app inside of the IIS worker process(w3wp.exe)
* OutOfProcess hosting model forward web requests to a backend ASP.NET Core app running the Kestrel Server

# Main Method in ASP.NET Core

* A Console application usually has a **Main()** method
* Why do we have a **Main()** method in ASP.NET Core web application
* ASP.NET Core application initially starts as a Console application and the **Main()** method of Program.cs class is the entry point



* This **Main()** method configures ASP.NET Core and starts it and at that point it becomes an ASP.NET Core web application

# ASP.NET Core InProcess Hosting

## Some of the Tasks that CreateDefaultBuilder() performs

* Setting up the web server
* Loading the host and application configuration from various configuration sources and
* Configuring logging

## An ASP.NET core application can be hosted

* InProcess or
* OutOfProcess

### To configure InProcess hosting



#### With InProcess

* Application is hosted inside the IIS worker process
* There is only one web server
* From a performance standpoint, InProcess hosting is better than OutOfProcess hosting



CreateDefaultBuilder() method calls UseIIS() method and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe)



ASP.NET application when run from CLI and **InProcess** hosting is configured than kestrel is used as the web server as shown below:





InProcess hosting delivers significantly higher request throughput than OutOfProcess hosting

### With OutOfProcess hosting

* 2 Web Servers – Internal and External Web Server
* The internal web server is Kestral
* The external web server can be IIS, Nginx, Apache

## What is Kestral

* Cross-Platform Web Server for ASP.NET Core
* Kestral can be used, by itself as an edge server
* The process used to host the app is dotnet.exe

# ASP.NET Core OutOfProcess Hosting

## To configure OutOfProcess hosting

We can either set AspNetCoreHostingModel to **OutOfProcess** or remove the AspNetCoreHostingModel, application uses OutOfProcess hosting by default



When application is run from CLI and OutOfProcess hosting is configured the Kestral server is used as edge server as shown below



Another way Kestral can be used in combination with a reverse proxy server, where Kestral is not facing the internet, it’s the Reverse Proxy server that takes the incoming http request and forwards it to the Kestral server that is hosting and running our ASP.NET Core application



A **Reverse Proxy Server**  such as IIS, Nginx etc. provides an additional layer of configuration and security, it might integrate better with our existing infrastructure and can also be used for load balancing.



# ASP.NET Core launchsettings.json File



# ASP.NET Core appsettings.json File

## Configurations Sources in ASP.Net Core

* Files(appsettings.json, appsettings.{Environment}.json)
* User secrets
* Environment variables
* Command-line arguments

Upon inspecting the file, you will see, the following is the default order in which the various configuration sources are read

1. appsettings.json,
2. appsettings.{Environment}.json
3. User secrets
4. Environment variables
5. Command-line arguments

# Middleware in ASP.NET Core

Middleware is a piece of software that can handle an HTTP request or response. A given middleware component in ASP.NET Core has a very specific purpose. For example we may have a middleware component that authenticates a user, another piece of middleware to handle errors, yet another middleware to serve static files such as JavaScript files, CSS files, Images etc.

It is these middleware components that we use to setup a request processing pipeline in ASP.NET Core. It is this pipeline that determines how a request is processed. The request pipeline is configured as part of the application startup by the Configure() method in Startup.cs file

## Middleware in ASP.NET Core

* Has access to both Request and Response
* May simply pass the Request to Middleware
* May process and then pass the Request to next Middleware
* May handle the Request and short-circuit the pipeline
* May process the outgoing Response
* Middlewares are executed in the order they are added

**Consider the following code in the Configure() method.**



**Code Explanation**

* We are using Run() method to add middleware to our application's request processing pipeline
* If you hover the mouse over the Run() method, from the intellisense you can see that this Run() method is implemented as an extension method of IApplicationBuilder interface. This is the reason we are able to invoke this Run() method on IApplicationBuilder object app.
* The parameter that we are passing to the Run() method is a RequestDelegate which we can see from the intellisense.
* RequestDelegate is a delegate that has HttpContext object as a parameter.
* It is through this HttpContext object, the middleware gains access to both the incoming http request and outgoing http response.
* At the moment, we are passing request delegate inline as an anonymous method using a lambda.
* Instead of passing the request delegate inline as an anonymous method, we can define the request delegate in a separate reusable class.
* With this Run() extension method we can only add a terminal middleware to the request pipeline.
* A terminal middleware is a middleware that does not call the next middleware in the pipeline

# Configure Request Processing Pipeline

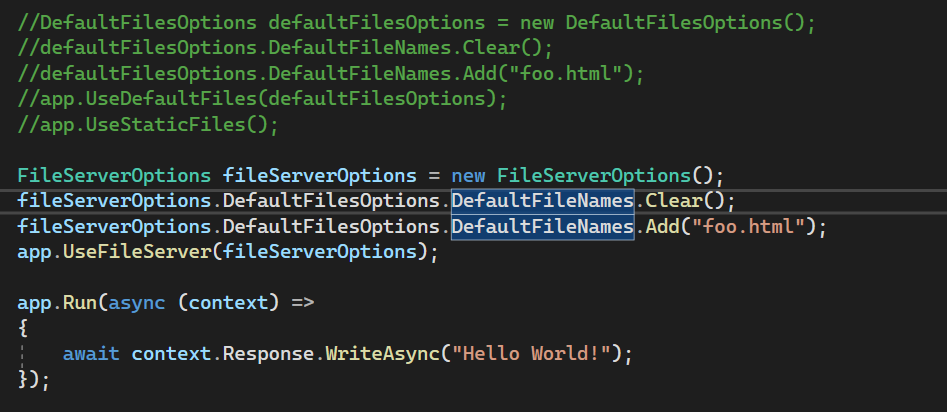




## So here are the 3 very important points to keep in mind regarding the request processing pipeline (see above image for reference)

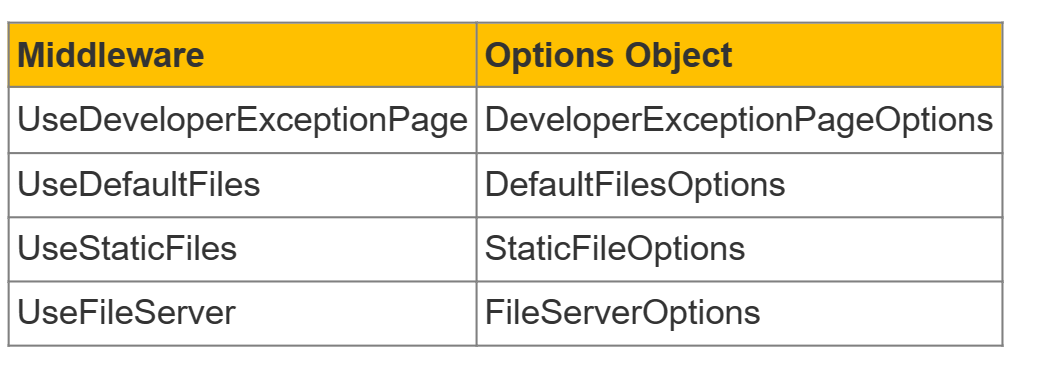
* Everything that happens before the next() method is invoked in each of the middleware components, happen as the request travels from middleware to middleware through the pipeline and this is represented by the **incoming arrow**.
* When a middleware handles the request and produces response, the request processing **pipeline starts to reverse**.
* Everything that happens after the next() method is invoked in a middleware component, happens as the response travels from middleware to middleware through the pipeline and this is represented by the **outgoing arrow**.

# Static Files in ASP.NET Core



The important point to keep in mind is the pattern that we use to add and customize these middleware components.

In most cases we add middleware components to our application’s request processing pipeline using **extension method** names that start with the word ‘USE’ for example: app.UseDeveloperExceptionPage(), app.UseDefaultFiles(), app.UseStaticFiles(), app.UseFileServer() and to customize these middleware components we use the respective ‘**Options’** Object for example: to customize ‘DeveloperExceptionPage’ middle we use ‘DeveloperExceptionPageOptions’ object, to customize ‘FileServer’ middleware we use ‘FileServerOptions’ object etc.



# ASP.NET Core Developer Exception Page



* To enable plug in **UseDeveloperExceptionPage** Middleware in the pipeline
* Must be plugged in the pipeline **as early as possible**
* Contains **Stack Trace, Query String, Cookies, HTTP headers** and **Routing**
* For customizing use **DeveloperExceptionPageOptions** Object

# ASP.NET Core Environment Variables

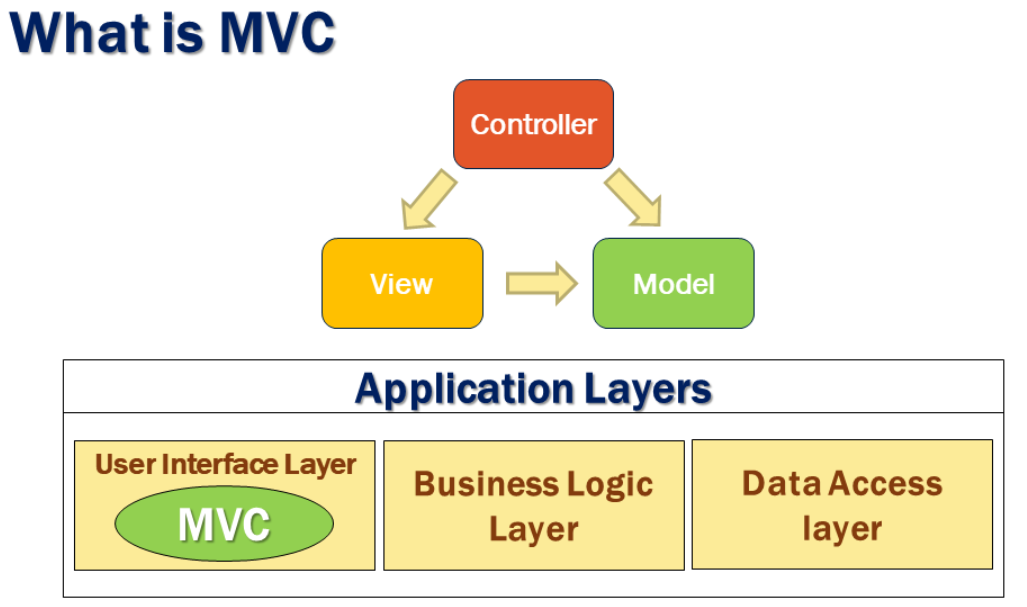


* ASPNETCORE\_ENVIRONMENT variables sets the **Runtime Environment**
* On development machine we set it in **launchsettings.json** file
* On Staging and Production server we set in the operating system
* Use **IHostingEnvironment** service to access the runtime environment
* Runtime environment defaults to **Production** if not set explicitly
* In addition to **standard environments (Development, Staging, Production),**

**Custom environments(UAT, QA etc)** are also supported

# ASP.NET Core MVC

## What is MVC ?



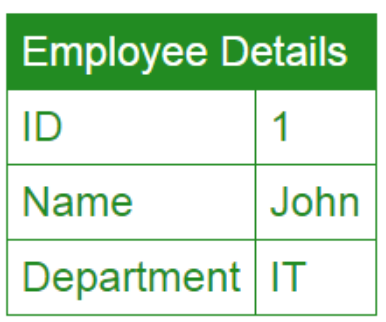
MVC consists of three fundamental parts - **Model**, **View**and **Controller**. It's an architectural design pattern for implementing User Interface Layer of an application. A typical real world application usually has the following layers.

* User Interface Layer
* Business Logic Layer or Domain Layer
* Data Access layer

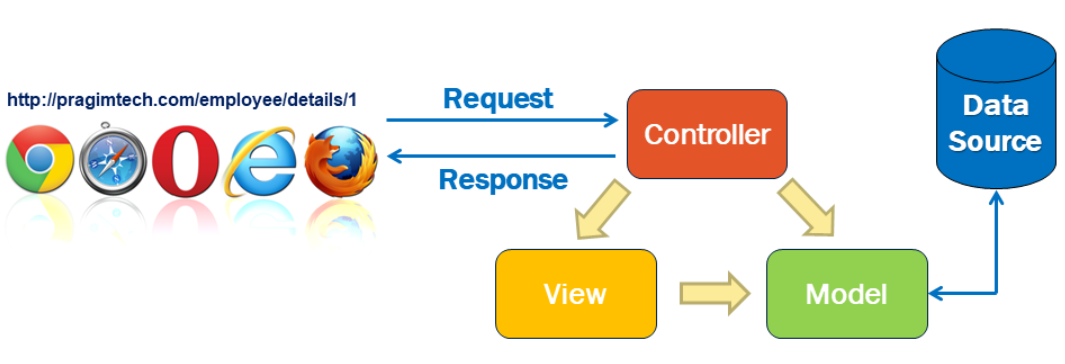
MVC design pattern is usually used for implementing the User Interface Layer of the application.

## How MVC Works ?

Let's say we want to retrieve a specific employee details (i.e. an employee whose ID is 1) and display those details on a web page in an HTML table as shown below.



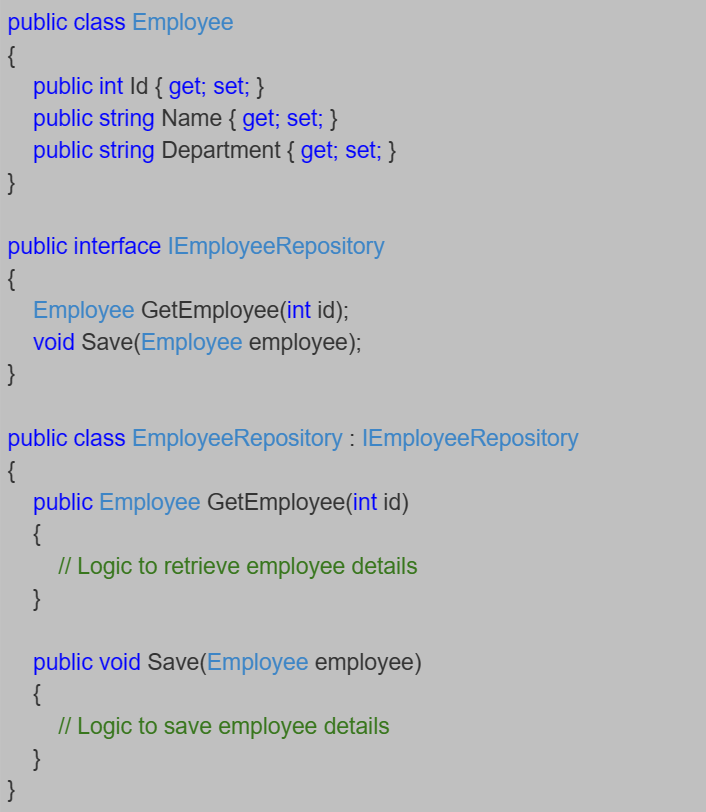
So from the web browser we issue a request and the URL may look something like the following  
[**http://pragimtech.com/employee/details/1**](http://pragimtech.com/employee/details/1)



1. When this request arrives at our server, it is the Controller in the MVC design pattern that receives the request and handles it.
2. The controller creates the model. The model has the classes that describe the data.
3. In addition to the data itself, Model also contains the logic to retrieve data from the underlying data source such as a database.
4. In addition to creating the Model, the controller also selects a View and passes the Model object to that View.
5. The view is only responsible for presenting the model data.
6. It is the view, that generates the required HTML to present the model data i.e the employee data provided to it by the Controller.
7. This HTML is then sent over the network to the user who made the request.

### Model

So model in this case consists of the Employee class + the EmployeeRepository class that manages the employee data as shown below.



We use the Employee class to hold the employee data and It is the EmployeeRepository class that retrieves and saves data in the underlying data source.   
  
So to generalise this, **a Model in MVC contains a set of classes that represent data and the logic to manage that data.** The class that represent the data is the Employee class and the class that manages the data is the EmployeeRepository class.  
  
If you are wondering why are we using the interface IEmployeeRepository. Can't we use just the EmployeeRepository class without the interface.  
  
Well, we can, but using the interface abstraction allows us to use dependency injection which in turn allows us to create systems that are loosely coupled  and easily testable.

### View

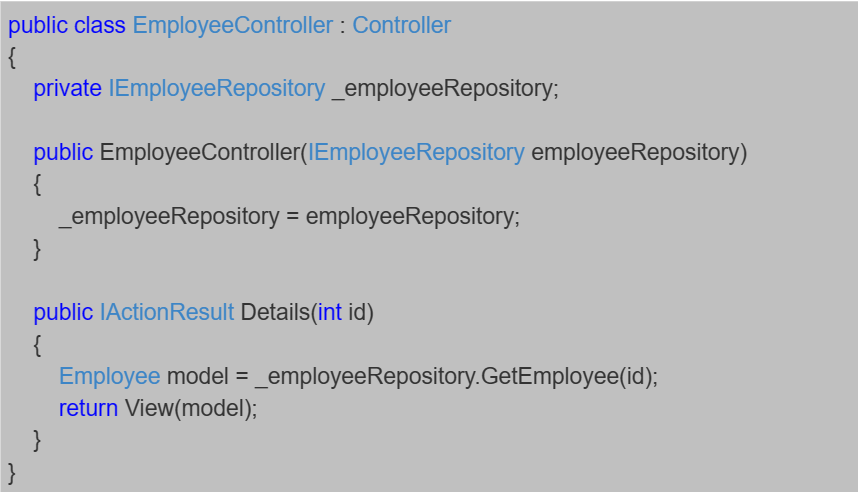
A View in MVC should only contain the logic to display the Model data provided to it by the Controller. You can think of a view as an HTML Template. Let's say in our example, we want to display Employee data in an HTML table.  
  
The view in this case will be provided with the Employee object. The Employee object is the model that carries the employee data to the view. The only responsibility of the view is to present the employee data in an HTML table. Here is the code in the view.



In MVC, a View in only responsible for presenting the model data. There should be no complex logic in a view. To maintain a clear separation of concerns, the logic in a view must be very minimal and that too it must only be there for presenting data. If you get to a point where the presentation logic is getting too complicated, consider using a **ViewModel**or **View Component**. **View Components**are new in this version of MVC.

### Controller

When a request from the browser arrives at our application, it is the controller in the MVC design pattern, that handles the incoming http request and responds to the user action.   
  
In this case the user has issued a request to the URL (/employee/details/1), so this request is mapped to the Details action method in the EmployeeController, passing it the EMPLOYEE ID which in this case is 1. This mapping is done by the **Routing rules**defined in our application.



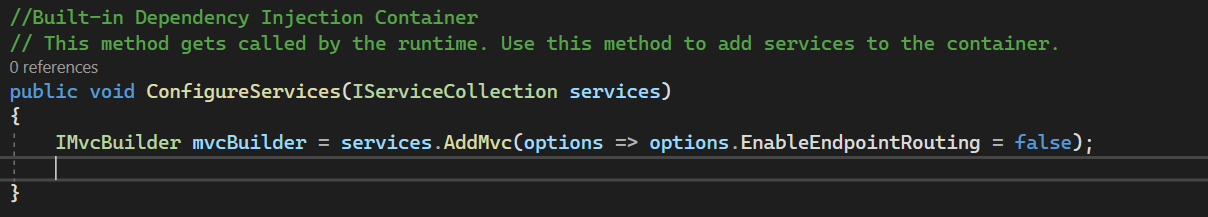
As you can see, from the code in the **Details**action method, the controller builds the model, in this case the Model is the **Employee**object. To retrieve the Employee data from the underlying data source, the controller is making use of the **EmployeeRepository**class.   
  
Once the controller has constructed the Employee model object with the required data, it then passes that Employee model object to the view. The view then generates the required HTML to present the Employee data provided to it by the Controller. This HTML is then sent over the network to the user who made the request.

## Summary

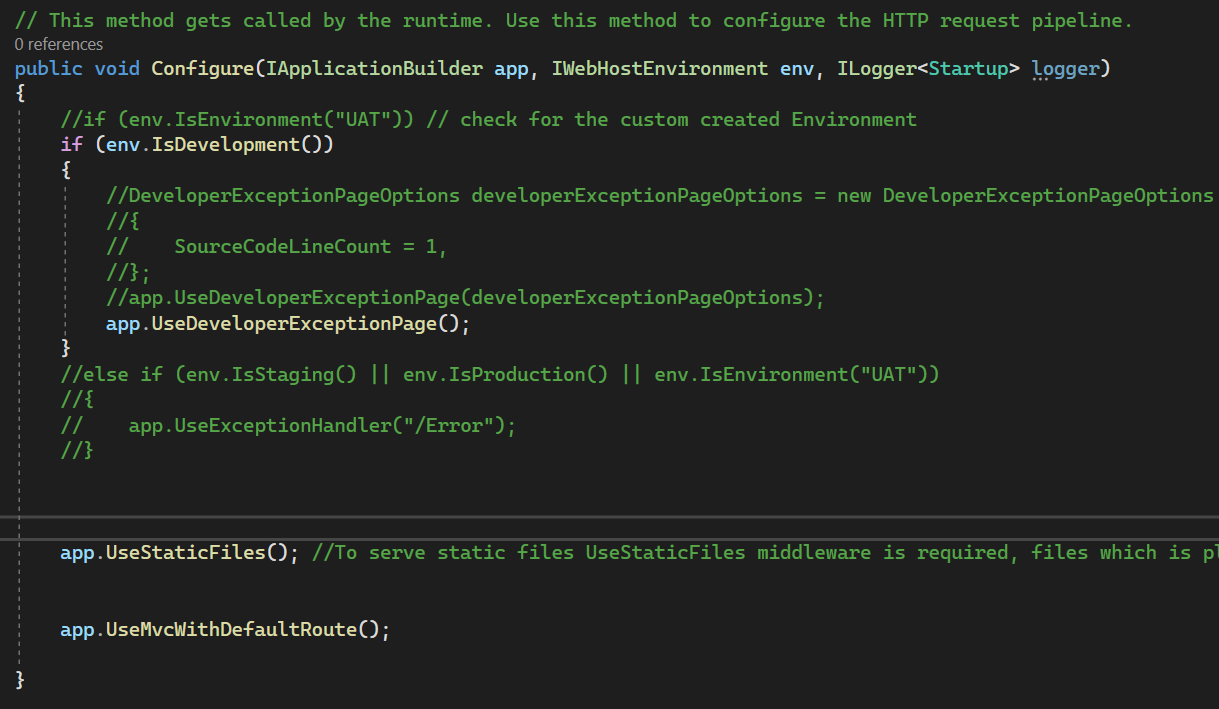
* MVC is an architectural design pattern for implementing User Interface Layer of an application
* **Model :** Set of classes that represent data + the logic to manage that data. Example - Employee class that represents the Employee data + EmployeeRepository class that saves and retrieves employee data from the underlying data source such as a Database.
* **View :** Contains the display logic to present the Model data provided to it by the Controller
* **Controller :** Handles the http request, work with the model, and selects a view to render that model.

# Setup MVC in ASP.Net Core

## Step 1 : Add the MVC Services to the Dependency injection Container



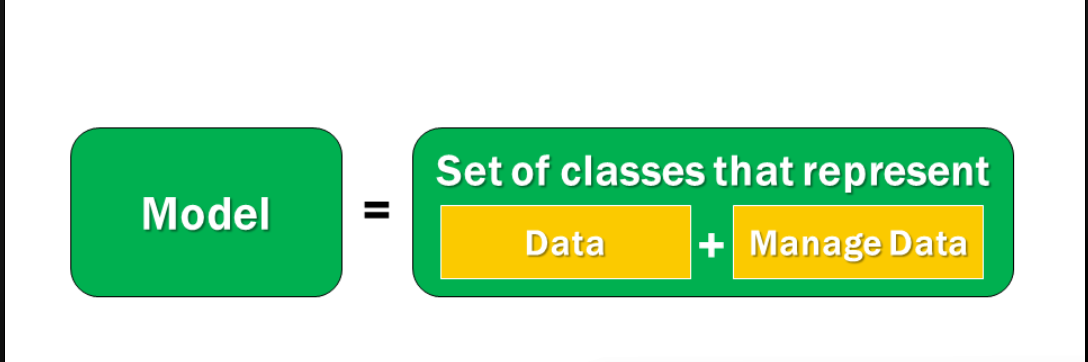
## Step 2: Add MVC middleware to Request Pipeline



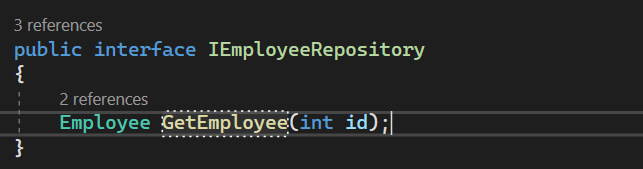
# ASP.NET Core AddMvc vs AddMvcCore

* AddMvcCore() method only adds the core MVC services
* AddMvc() method adds all the required MVC Services
* AddMvc() method internally calls the AddMvcCore() method.

# Model in ASP.Net Core



## MVC Model



# ASP.Net Core Dependency Injection

## Registering Services with the ASP.NET Core Dependency Injection Container

The method that we use determines the**lifetime of the registered service**.

* **AddSingleton()** - As the name implies, AddSingleton() method creates a **Singleton service**. A Singleton service is created when it is first requested. This same instance is then used by all the subsequent requests. So in general, a Singleton service is created only one time per application and that single instance is used throughout the application life time.
* **AddTransient()** - This method creates a **Transient service**. A new instance of a Transient service is created each time it is requested.
* **AddScoped()** - This method creates a **Scoped service**. A new instance of a Scoped service is created once per request within the scope. For example, in a web application it creates 1 instance per each http request but uses the same instance in the other calls within that same web request.