Chapter 13: Creating our first page

Gill Cleeren

@gillcleeren

Agenda

- The MVC pattern – recap
- Model, repositories and a controller please
- The view
- Styling things

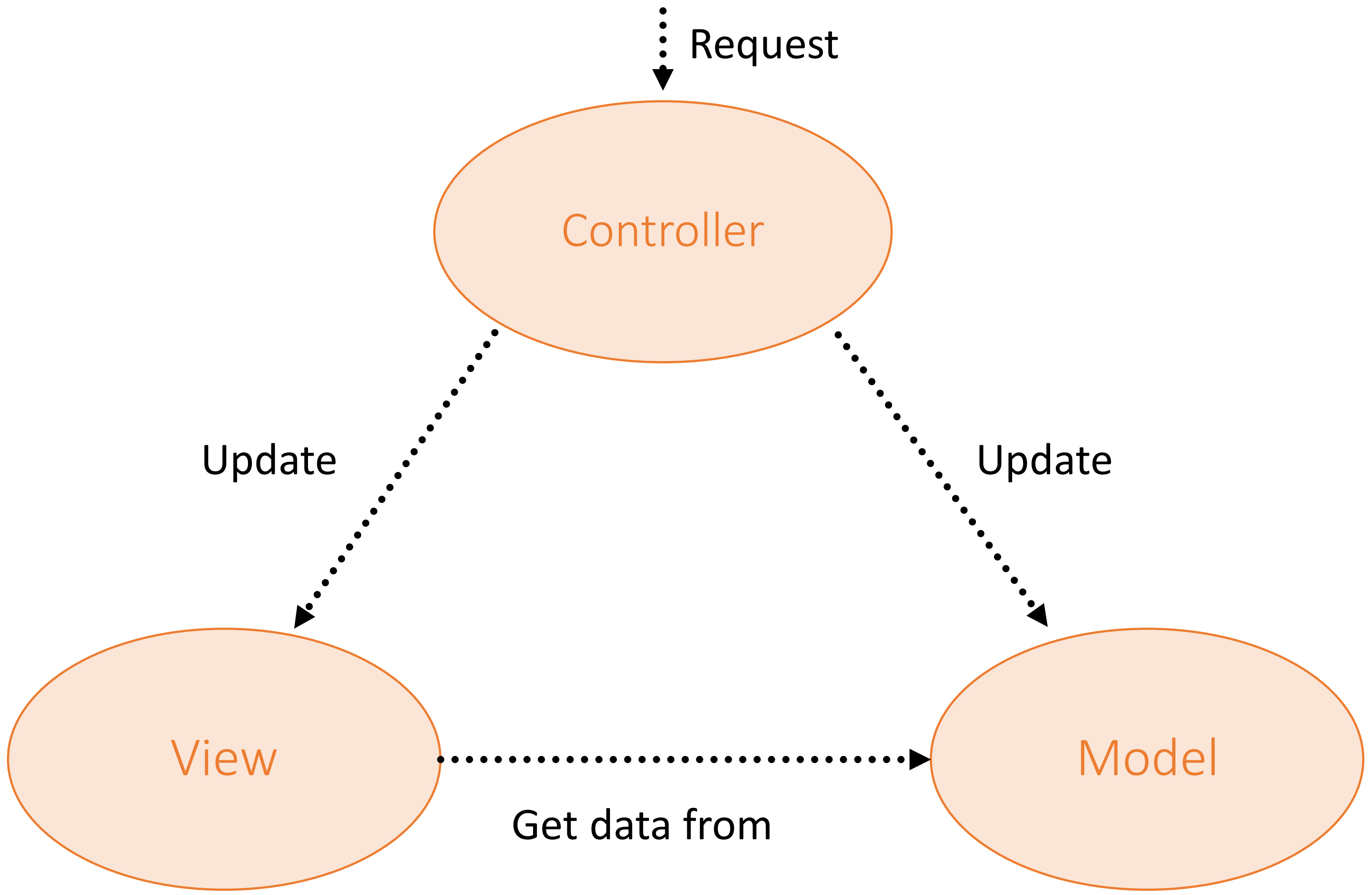
The MVC pattern – recap

The MVC in ASP.NET Core MVC

- Model-View-Controller
  - Architectural pattern
  - Separation of concerns
  - Promotes testability and maintainability

SNOWBALL

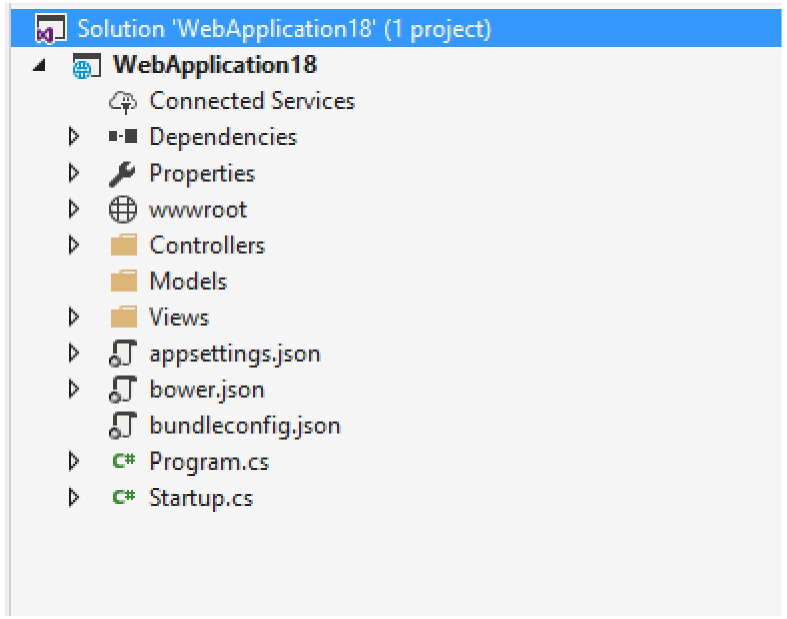
The MVC in ASP.NET Core MVC



Model, repositories and a controller

Convention-based folders

- Controllers
- Models
- Views



Solution 'WebApplication18' (1 project)
WebApplication18
Connected Services
Dependencies
Properties
wwwroot
Controllers
Models
Views
appsettings.json
bower.json
bundleconfig.json
Program.cs
Startup.cs

SNOWBALL

The M of Model

- Domain data
- Logic for maintaining the data
- Simple API
- Hide details of data management from consumer

Simple Model class

```
public class Pie
{
    public int PieId { get; set; }
    public string Name { get; set; }
    public string ShortDescription { get; set; }
    public decimal Price { get; set; }
    public int CategoryId { get; set; }
    public virtual Category Category { get; set; }
}
```

The repository pattern

- *It queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source*
- *A **repository** separates the business logic from the interactions with the underlying data source or Web service*
- *Contract based (in our approach)*

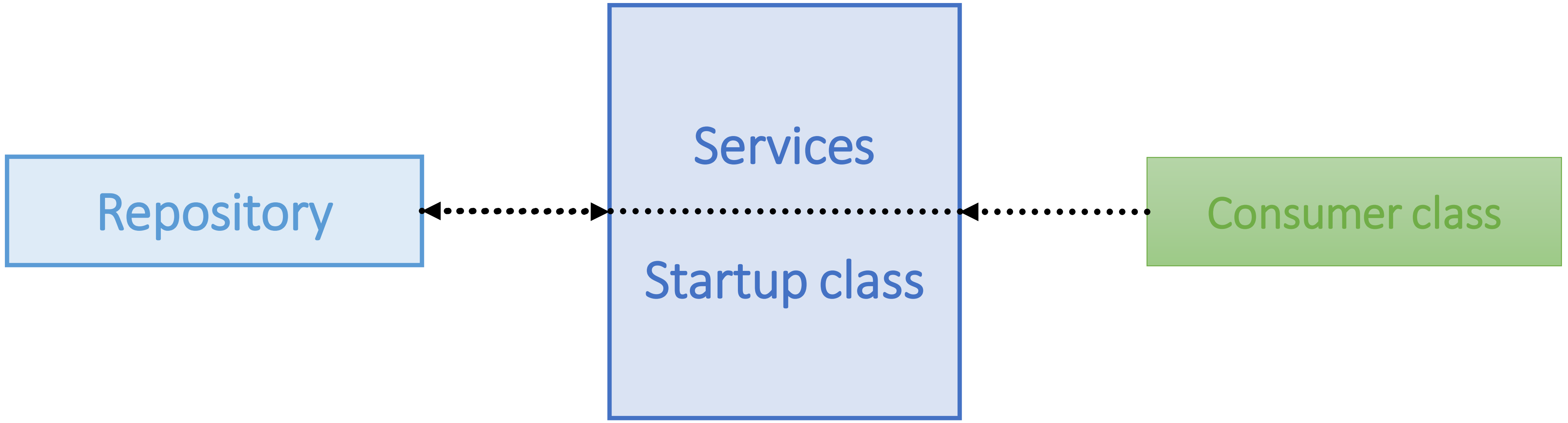
Sample repository

```
public interface IPieRepository
{
    IEnumerable<Pie> Pies { get; }
    IEnumerable<Pie> PiesOfTheWeek { get; }

    Pie GetPieById(int pieId);
}
```

Using mocks

```
public class MockPieRepository : IPieRepository
{
    public IEnumerable<Pie> Pies
    {
        get
        { ... }
    }

    public IEnumerable<Pie> PiesOfTheWeek
    {
        get
        { ... }
    }

    public Pie GetPieById(int pieId)
    { ... }
}
```

Registering the repository



Registration options in ASP.NET Core

- AddTransient
  - Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services
- AddSingleton
  - Singleton lifetime services are created the first time they are requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance
- AddScoped
  - Scoped lifetime services are created once per request

Registering the repository

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IPieRepository, MockPieRepository>();
    services.AddMvc();
}
```

Other sample

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase()
    );

    // Add framework services.
    services.AddMvc();

    // Register application services.
    services.AddScoped<ICharacterRepository, CharacterRepository>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
    services.AddTransient<OperationService, OperationService>();
}
```

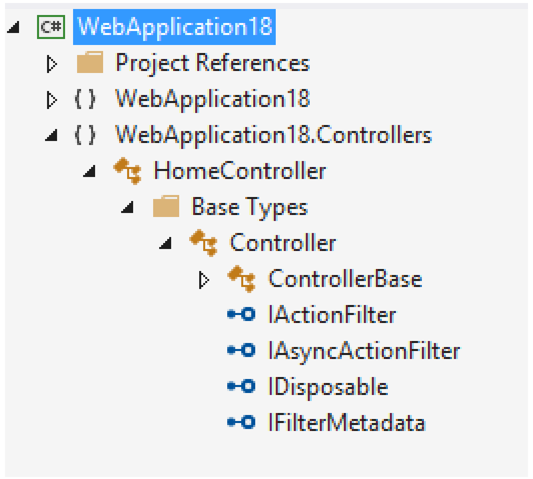
DEMO

Taking a look at the model & repository

Registration in the Startup

The C of Controller

- Handle user request
- Build and update the model
- No knowledge about data persistence
- Return the result for a view
- Class that inherits from base Controller class
- In .NET Core, a controller can be for API and web
  - Same base class
  - We'll see API's later in this course!



```
WebApplication18
  Project References
  WebApplication18
  WebApplication18.Controllers
    HomeController
      Base Types
        Controller
          ControllerBase
          IActionFilter
          IAsyncActionFilter
          IDisposable
          IFilterMetadata
```

A very first, basic controller

- Action
- View

```
public class PieController : Controller
{
        public ViewResult Index()
        {
                return View();
        }
}
```

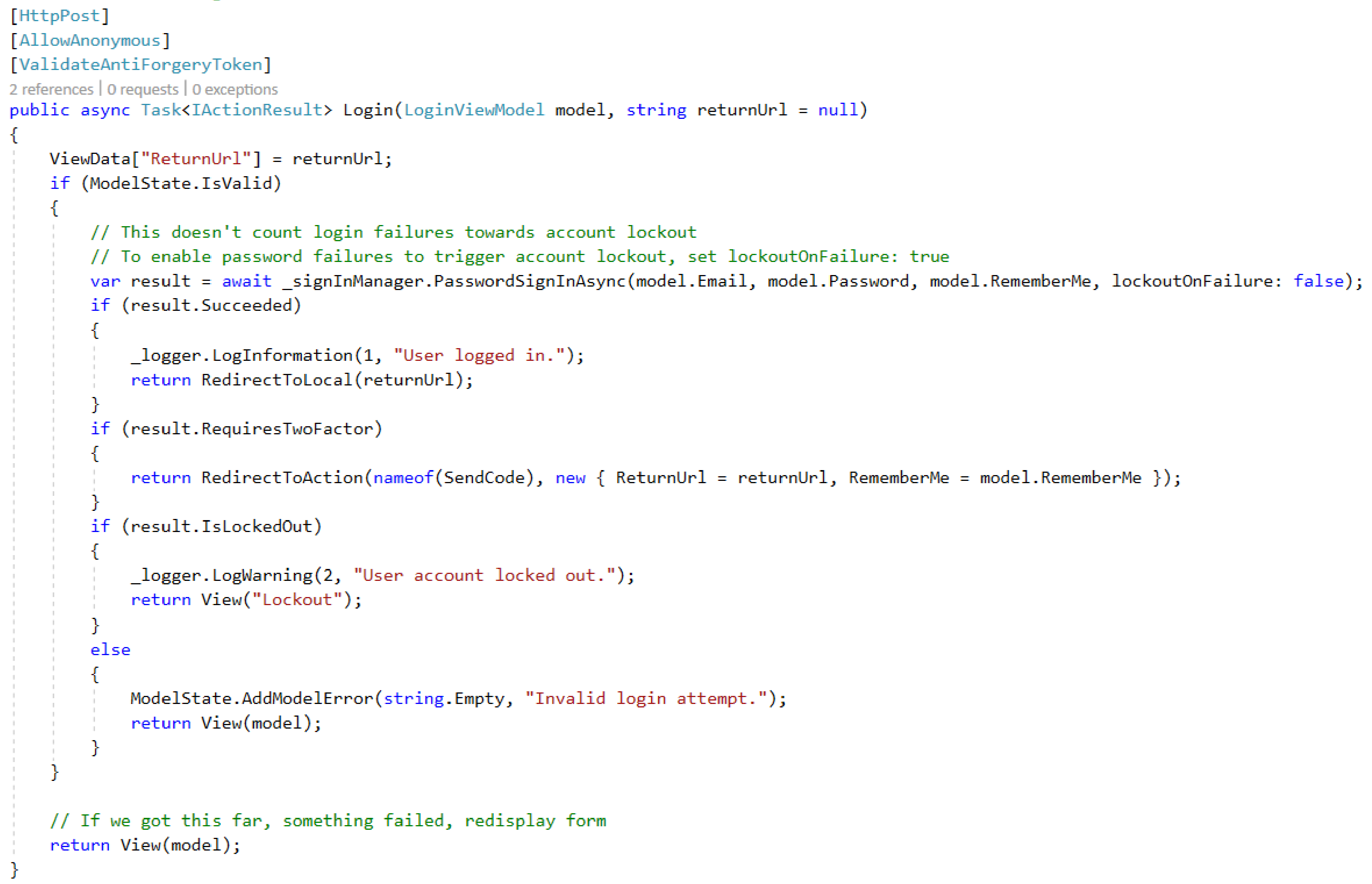
A real controller

```
public class PieController : Controller
{
    private readonly IPieRepository _pieRepository;

    public PieController(IPieRepository pieRepository)
    {
        _pieRepository = pieRepository;
    }

    public ViewResult List()
    {
        return View(_pieRepository.Pies);
    }
}
```

The A of Action (oops ☺)

- Invokable on the Controller
- Public method
  - Can’t be static
  - Can’t be extension method
  - Can’t be constructor
  - Can’t be getter/setter
  - Can’t have open generic type
  - Can’t be a method of base Controller class
  - Can’t have ref or out parameters

May get complex, not always great

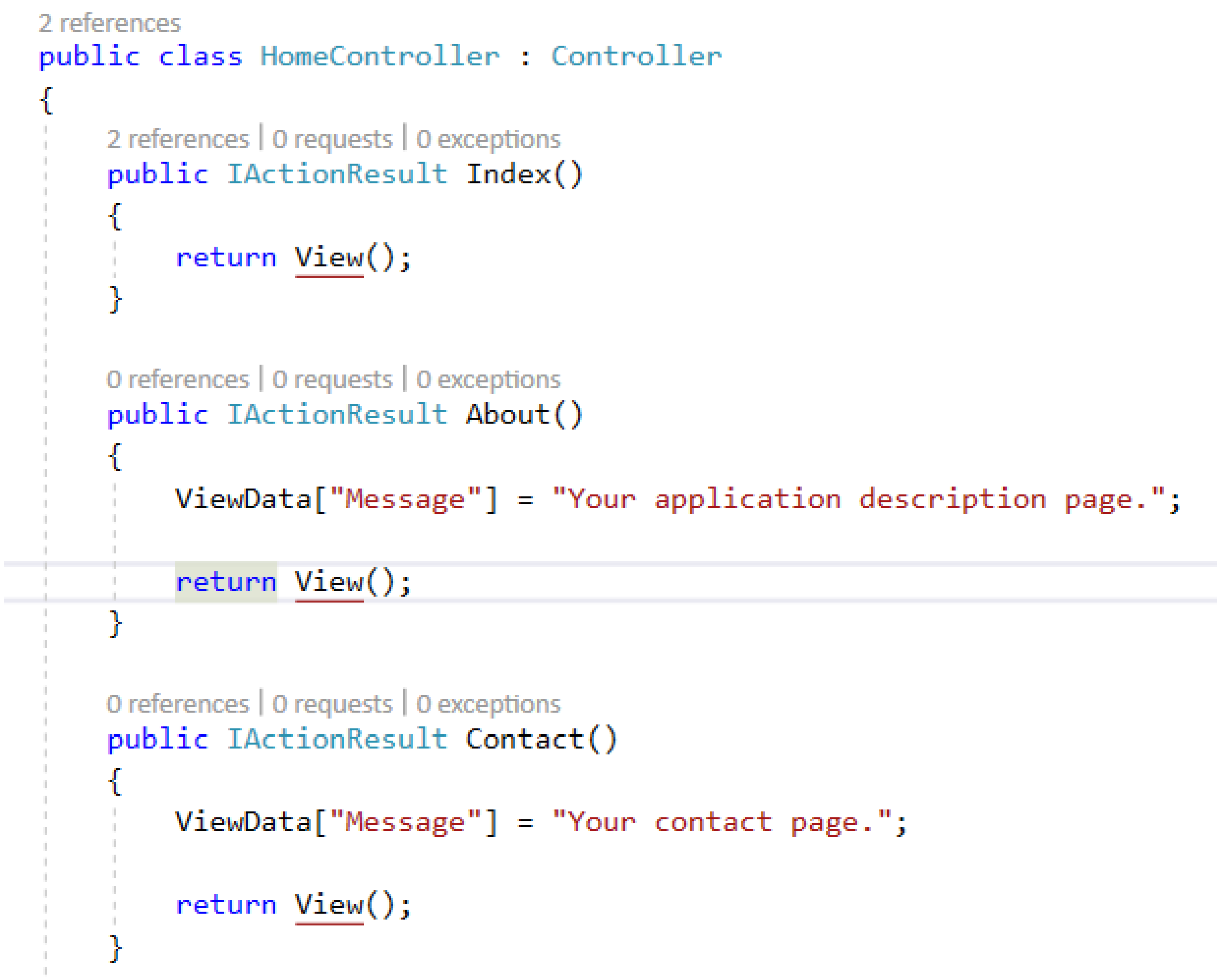


```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
2 references | 0 requests | 0 exceptions
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe = model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Actions return ActionResults

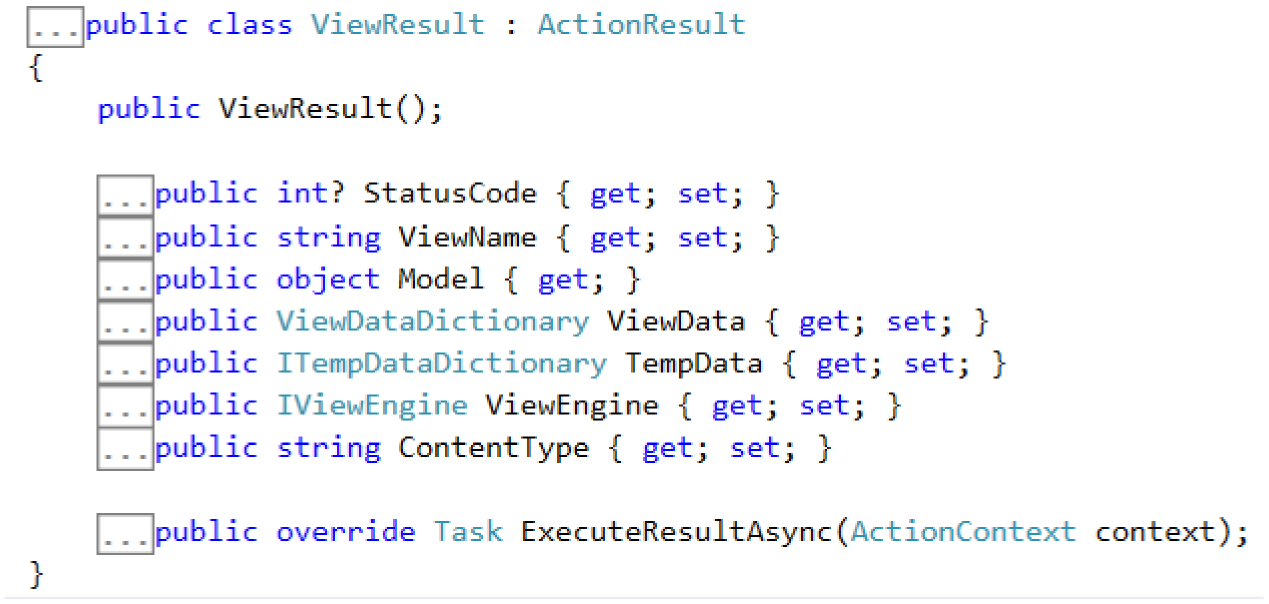
- Is the result of the execution of the Action
- Basically instructs what is next to be done
- ASP.NET Core comes with 30+ built-in ActionResults
  - MVC: View, Redirect
  - WebAPI: status-based, Json
  - Files: to download files to the client
- Implement IActionResult interface

Actions return ActionResults



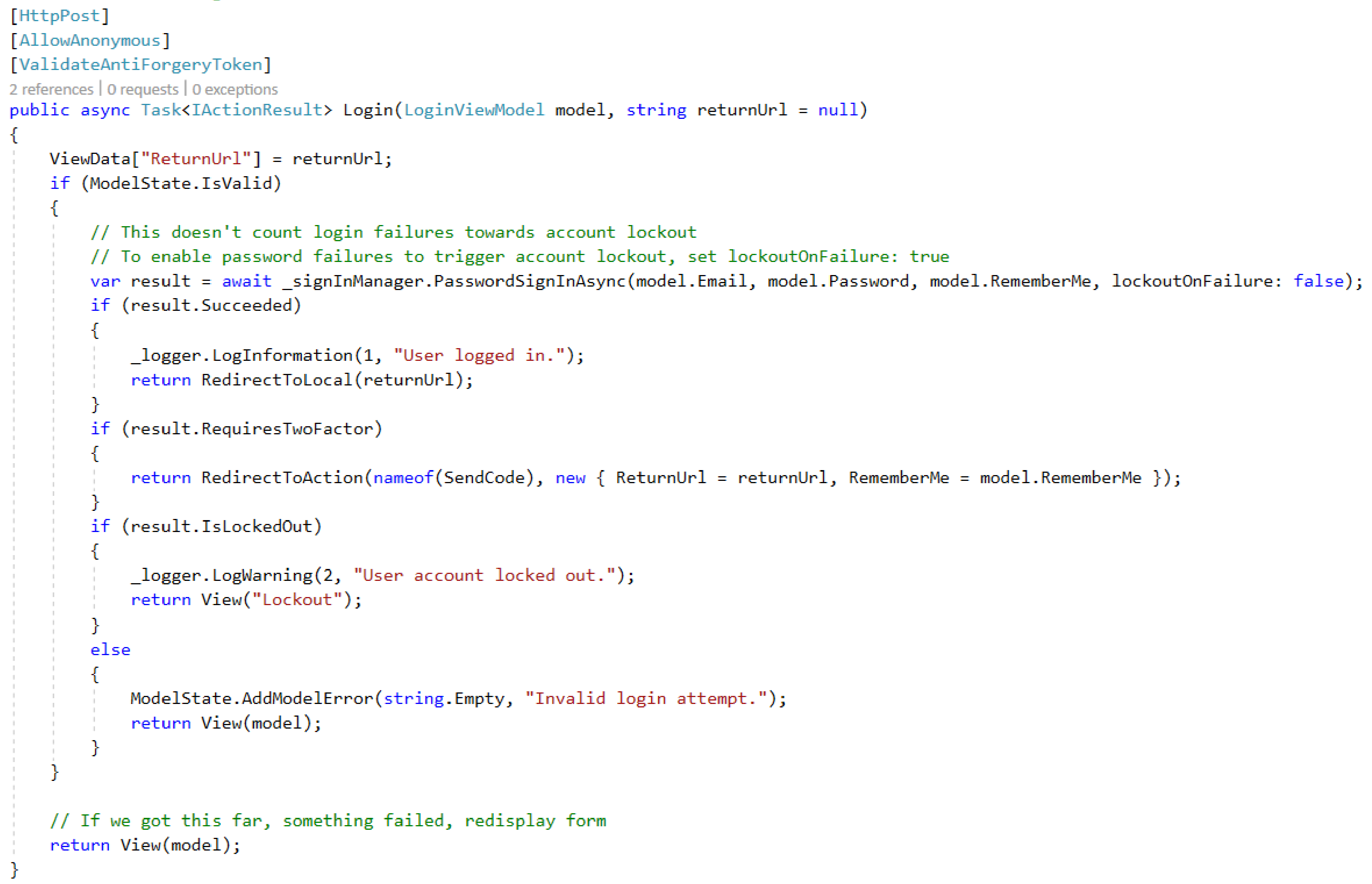
```
2 references
public class HomeController : Controller
{
    2 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }

    0 references | 0 requests | 0 exceptions
    public IActionResult About()
    {
        ViewData["Message"] = "Your application description page.";

        return View();
    }

    0 references | 0 requests | 0 exceptions
    public IActionResult Contact()
    {
        ViewData["Message"] = "Your contact page.";

        return View();
    }
```

ViewResult



```
[...]public class ViewResult : ActionResult
{
    public ViewResult();

    [...]public int? StatusCode { get; set; }
    [...]public string ViewName { get; set; }
    [...]public object Model { get; }
    [...]public ViewDataDictionary ViewData { get; set; }
    [...]public ITempDataDictionary TempData { get; set; }
    [...]public IViewEngine ViewEngine { get; set; }
    [...]public string ContentType { get; set; }

    [...]public override Task ExecuteResultAsync(ActionContext context);
}
```

Returning multiple IActionResults



```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
2 references | 0 requests | 0 exceptions
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe = model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Typical MVC Action Results

- ViewResult
- RedirectToActionResult & RedirectToRouteResult
- RedirectResult
- JsonResult
- PartialViewResult
- And many more

Typical API Action Results

- OkResult
- CreatedResult
- BadRequestResult
- UnauthorizedResult
- NotFoundResult
- NoContentResult

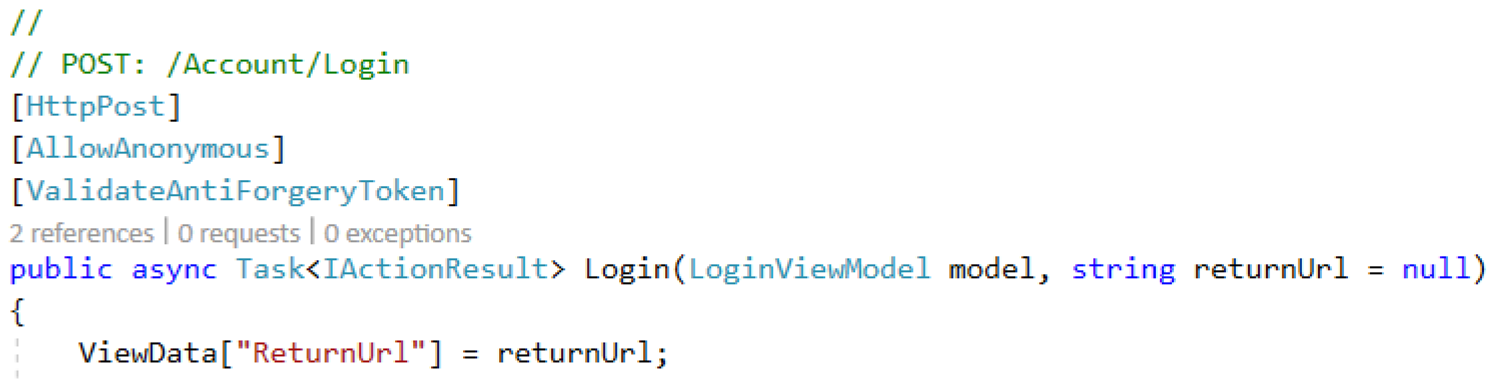
File Result

- ContentResult
  - Raw string
- PhysicalFileResult
  - Physical path-based file
- FileContentResult
  - Byte array
- FileStreamResult
  - Stream content

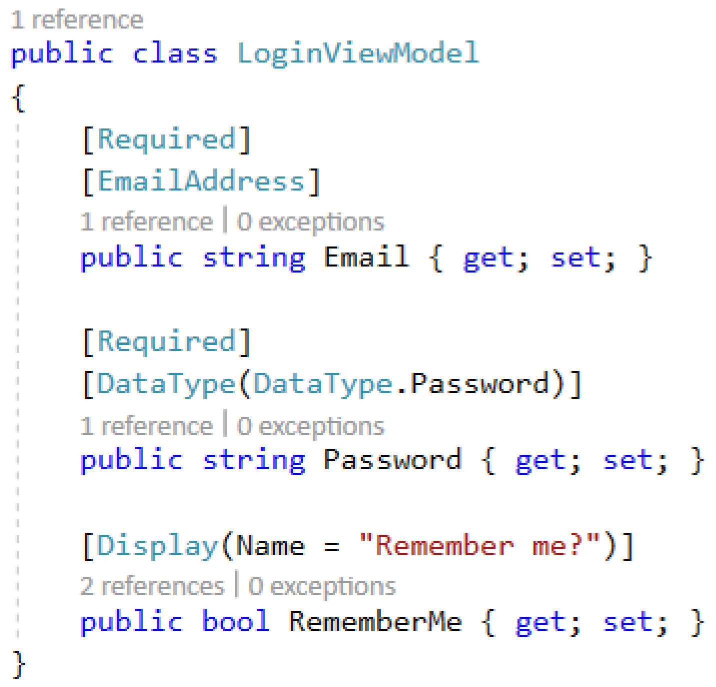
Model binding

- Will transform request variables in action parameters
  - Passing a Pie instance
- Will search for required properties
  - Model binders
    - Form data
    - Query string
    - Route values
    - More can be added if needed

Model binding



```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
2 references | 0 requests | 0 exceptions
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
```



```
1 reference
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    1 reference | 0 exceptions
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    1 reference | 0 exceptions
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    2 references | 0 exceptions
    public bool RememberMe { get; set; }
}
```

DEMO

Taking a look at the Controller

The V of View

- Can be “regular” view or a strongly-typed view
- Must be kept as dumb as possible
  - Difficult to test
- Limit conditional & nested logic in view if possible
- Made “smart” using Tag Helpers and HTML Helpers
  - More on these later

Regular view

```
<!DOCTYPE html>

<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <div>
         Welcome to Bethany’s Pie Shop
    </div>
  </body>
</html>
```

SNOWBALL

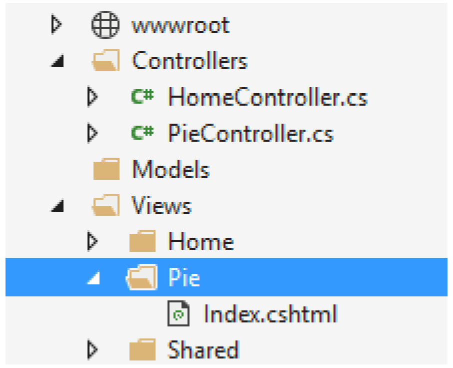
How do we know which view gets called?

```
public class PieController : Controller
{
        public ViewResult Index()
        {
                return View();
        }
}
```

Action

View to show

View Folder Structure



```
▷ wwwroot
◢ Controllers
    ▷ HomeController.cs
    ▷ PieController.cs
  Models
◢ Views
    ▷ Home
    ◢ Pie
        Index.cshtml
    ▷ Shared
```

Using the ViewBag

```
public class PieController : Controller
{
        public ViewResult Index()
        {
           ViewBag.Message = "Welcome to Bethany’s Pie Shop";
           return View();
        }
}
```

Making the UI somewhat dynamic

```
<!DOCTYPE html>

<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <div>
      @ViewBag.Message
    </div>
  </body>
</html>
```

That won’t get us very far…

Razor is a markup syntax which allows us to include C# functionality in our web pages

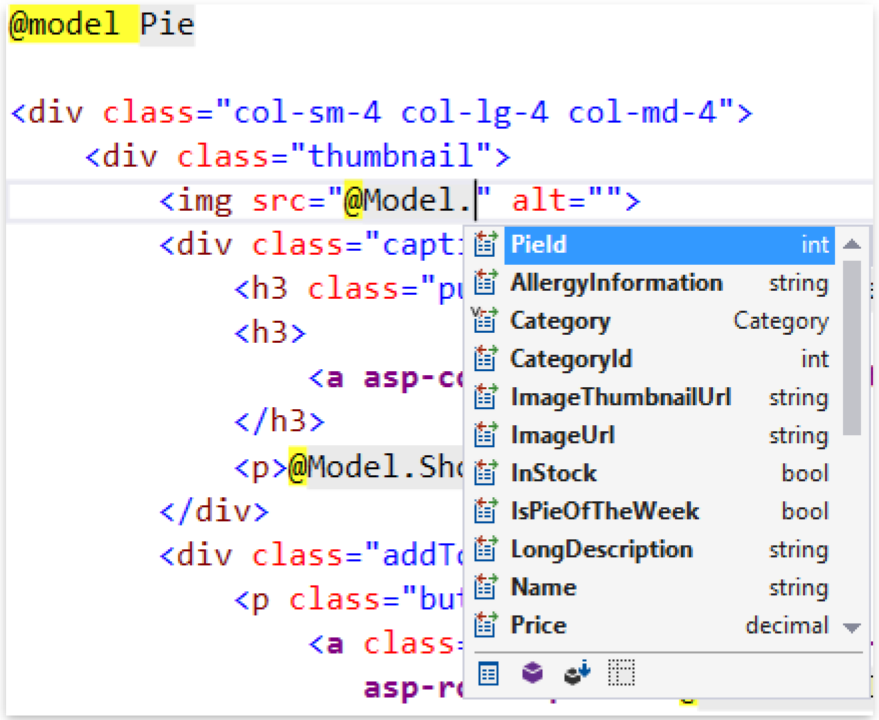
Making the view strongly-typed

```
public class PieController : Controller
{
    public ViewResult List()
    {
        return View(_pieRepository.Pies);
    }
}
```

Making the view strongly-typed

```
@model IEnumerable<Pie>
<html>
…
  <body>
    <div>
      @foreach (var pie in Model.Pies)
      {
        <div>
          <h2>@pie.Name</h2>
          <h3>@pie.Price.ToString("c")</h3>
          <h4>@pie.Category.CategoryName</h4>
        </div>
      }
    </div>
  </body>
</html>
```

Hello IntelliSense



```
@model Pie

<div class="col-sm-4 col-lg-4 col-md-4">
    <div class="thumbnail">
        <img src="@Model.|" alt="">
        <div class="capt
            <h3 class="pu
            <h3>
                <a asp-co
            </h3>
            <p>@Model.Sho
        </div>
        <div class="addTo
            <p class="but
                <a class=
                    asp-ro
```

```
PieId                 int
AllergyInformation    string
Category              Category
CategoryId            int
ImageThumbnailUrl     string
ImageUrl              string
InStock               bool
IsPieOfTheWeek        bool
LongDescription       string
Name                  string
Price                 decimal
```

SNOWBALL

DEMO

Taking a look at the View

Without a ViewModel, it might get hard

```
public class PiesListViewModel
{
    public IEnumerable<Pie> Pies { get; set; }
    public string CurrentCategory { get; set; }
}
```

DEMO

Looking at using a View Model

Layout with _Layout

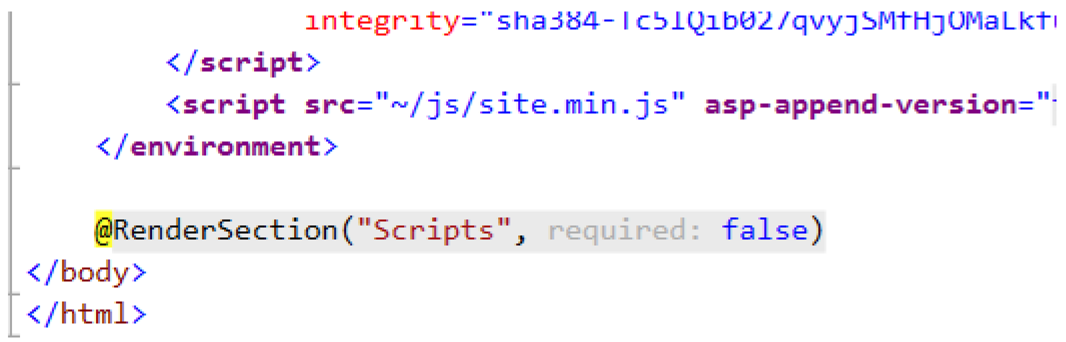
- Making it possible to use a template
- Lives in the Shared folder by default
  - ASP.NET Core will search for it in this location
- More than one are possible
  - Views can refer to specific one
  - A layout file can also be created in a different folder
- Will contain typically one or more placeholders
- By default, we will need to repeat it for every view…
  - Not really, just hold on a second!

_Layout example

- @RenderBody will be replaced with the actual view

```
<!DOCTYPE html>
<html>
        <head>
                <title>Bethany's Pie Shop</title>
        </head>
        <body>
                <div>
                        @RenderBody()
                </div>
        </body>
</html>
```

More than one section



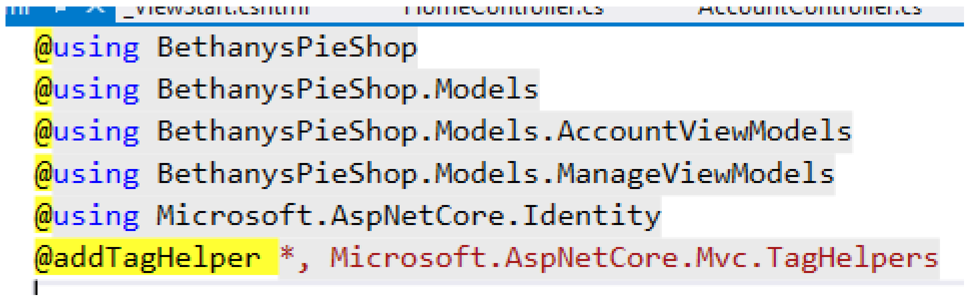
```
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkf
            </script>
            <script src="~/js/site.min.js" asp-append-version="
        </environment>

        @RenderSection("Scripts", required: false)
</body>
</html>
```

Using a _ViewStart.cshtml

- Will be searched for by default by ASP.NET Core when a view is rendered

```
@{
    Layout = "_Layout";
}
```

One more file: View Imports



```
@using BethanysPieShop
@using BethanysPieShop.Models
@using BethanysPieShop.Models.AccountViewModels
@using BethanysPieShop.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

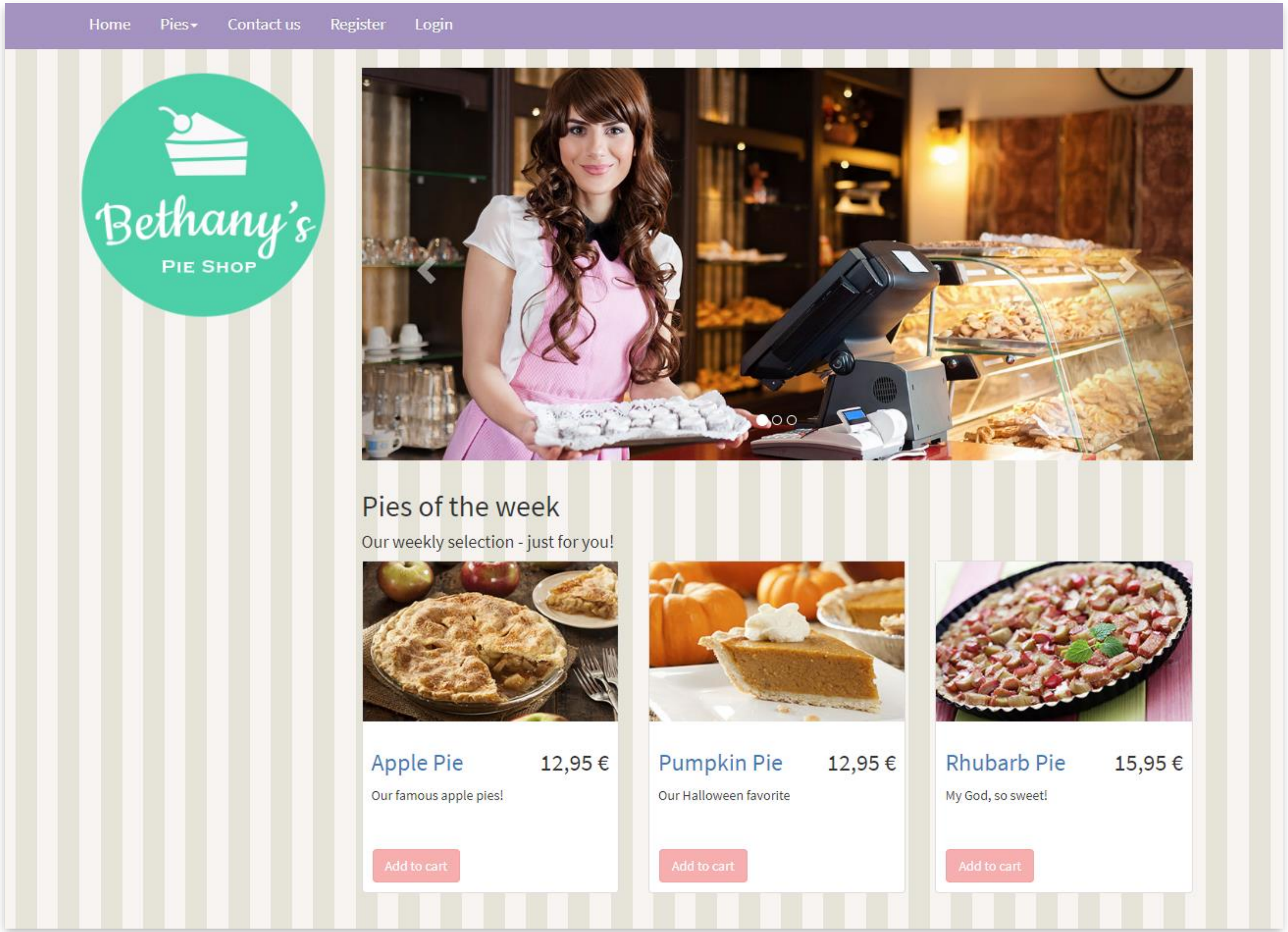
- More on this one later!

DEMO

Looking at the view files

Adding some style

Where We Need to Get



Pies of the week

Our weekly selection - just for you!

Apple Pie 12,95 €

Our famous apple pies!

Pumpkin Pie 12,95 €

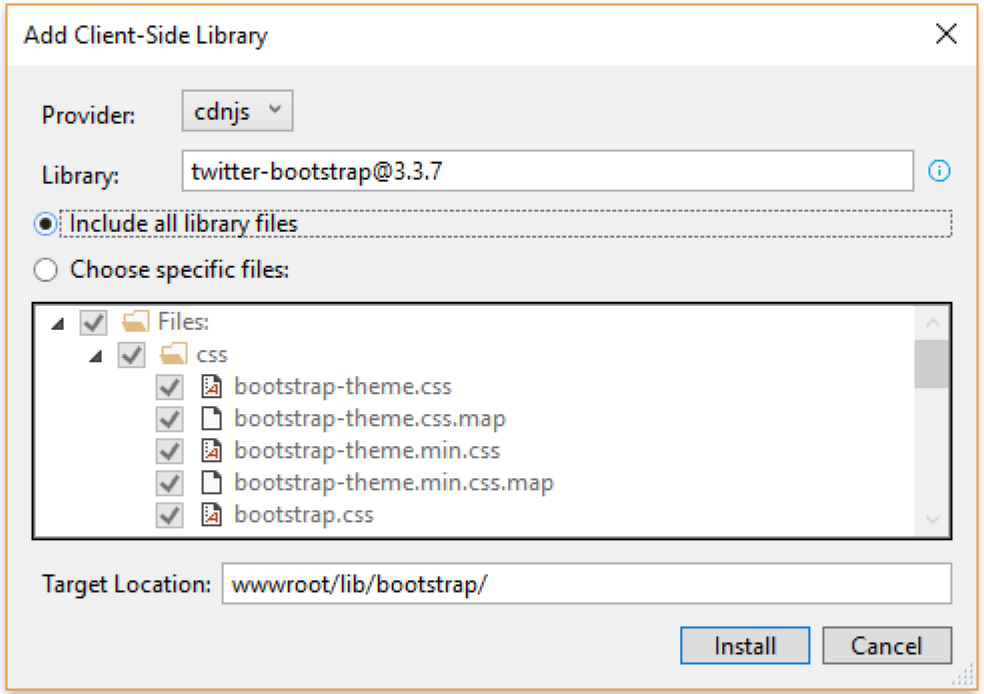
Our Halloween favorite

Rhubarb Pie 15,95 €

My God, so sweet!

Using Library Manager (LibMan)

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "twitter-bootstrap@3.3.7",
      "destination": "wwwroot/lib/bootstrap/"
    }
  ]
}
```


Add Client-Side Library

Provider: cdnjs

Library: twitter-bootstrap@3.3.7

(●) Include all library files

( ) Choose specific files:

```
Files:
  css
    bootstrap-theme.css
    bootstrap-theme.css.map
    bootstrap-theme.min.css
    bootstrap-theme.min.css.map
    bootstrap.css
```

Target Location: wwwroot/lib/bootstrap/

SNOWBALL

DEMO

Adding styles with Bootstrap

Summary

- First page created
  - M
  - V
  - C
- Client-side package management using LibMan

LAB

Do exercises 1 - 4