



.NET Core: Developing Cross-Platform Web Apps with ASP.NET Core – Workshop*PLUS*

< Engineer Name >

Customer Engineer

v3.1

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

© 2020 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at

<http://www.microsoft.com/en-us/legal/intellectualproperty/permissions/default.aspx>

Internet Explorer, Microsoft, Microsoft Corporate Logo, MSDN, Visual Studio, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Module 4: Controllers

Module Overview

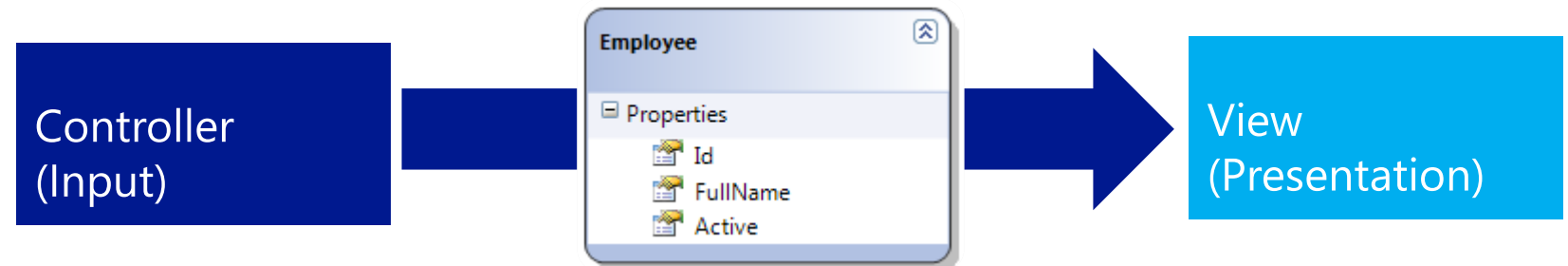
Module 4: Controllers

Section 1: Controller Fundamentals

Lesson: Role of Controllers

Role of Controllers

- Controllers are the initial entry point for a given request
- Responsible for:
 - Handling and responding to user input and interaction
 - Selecting which model types to work with
 - Selecting which view to render in response



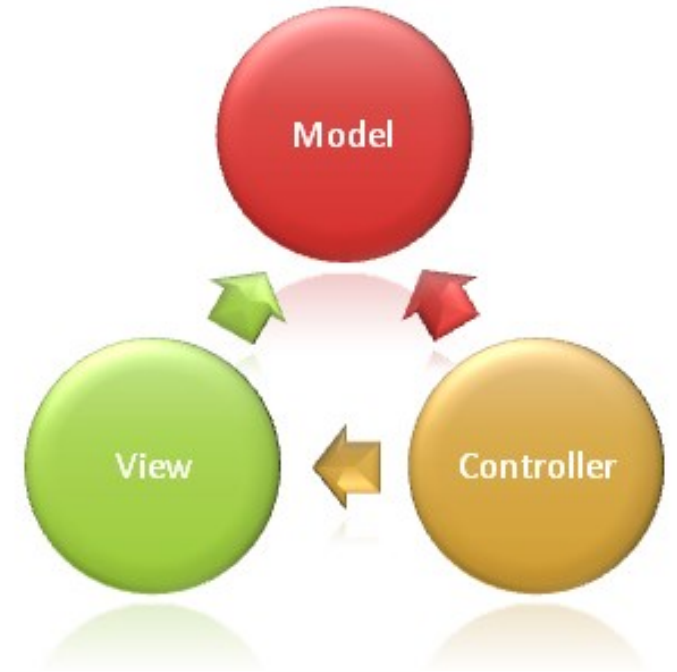
Role of Controllers (cont)

- Controllers are **classes** and are used to define and group a set of **actions**.
- **Actions** are methods on a controller which handle requests.
- Controllers logically group similar actions or functionality together.
- Allows common set of rules, such as routing, caching and authorization to be applied collectively.

Role of Controller Example

- Example

1. User sends a URL request with query string values
2. *Controller* is triggered against the request
3. *Controller* handles query-string values
4. *Controller* passes the values to the model
5. Model uses the value to query the database and returns the results
6. *Controller* selects a View to render the UI
7. *Controller* returns the View to the requesting browser



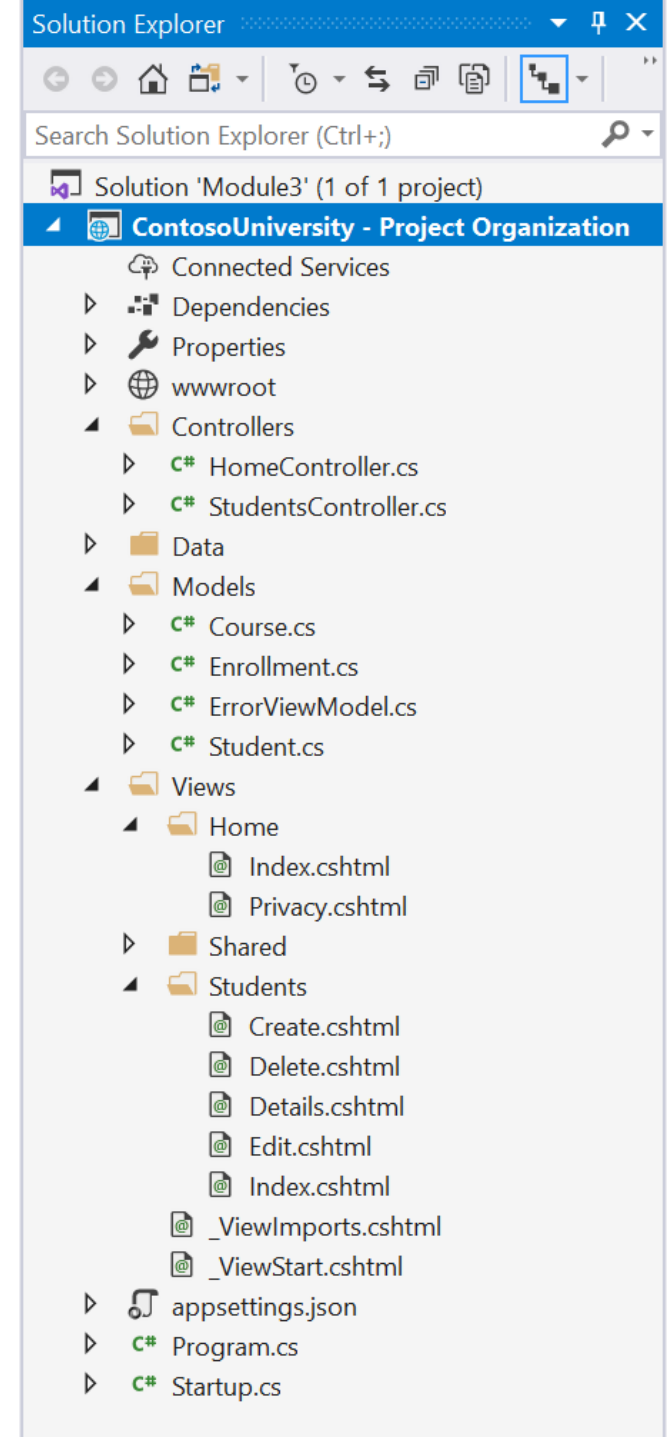
Module 4: Controllers

Section 2: Developing Controllers

Lesson: Basics

ASP.NET Core MVC Project File Organization

- Much of ASP.NET Core MVC is “by convention”
- Controller Conventions:
 - Reside in project’s Controllers folder
 - Suffixed by Controller (i.e. HomeController.cs)
 - Inherit from Microsoft.AspNetCore.Mvc.Controller
- ASP.NET Core MVC does not require this structure
- Convention over configuration
 - Default directory structure keeps application concerns clean
 - For example: Allows you to omit location paths when referencing views
 - By default, ASP.NET Core MVC looks for the View files in
\Views\[ControllerName]



Controller Development

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

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

        return View();
    }
}
```

Executing Actions

- `http://contoso.com/<controllername>/<actionname>`
- `http://contoso.com/home/Index`
- `http://contoso.com/home/About`

```
5 references
public class HomeController : Controller
{
    4 references
    public IActionResult Index()
    {
        // Index action implementation...
        return View();
    }

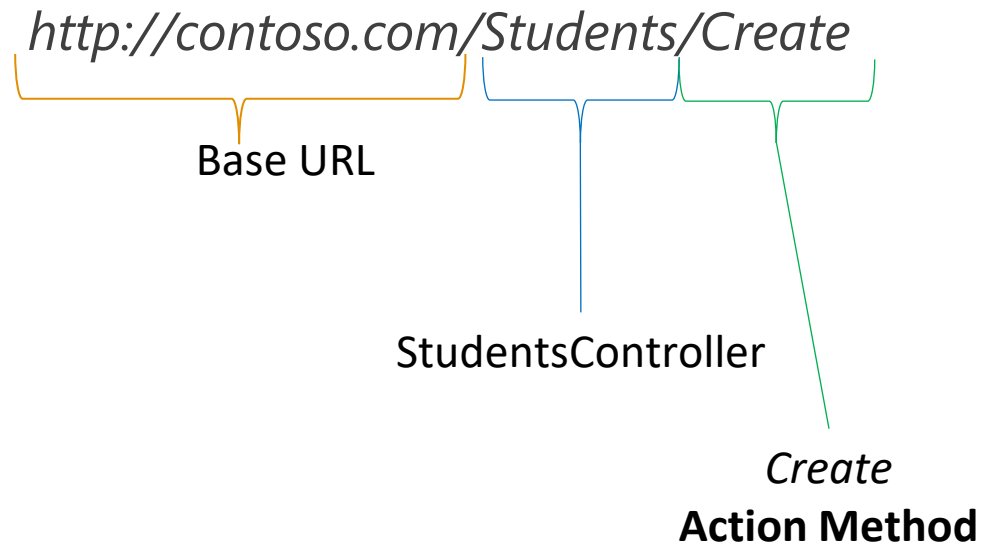
    0 references
    public IActionResult About()
    {
        ViewData["Message"] = "Your application description page.";

        return View();
    }
}
```

Action Methods

- Action (Action Method) are methods on a controller which handle requests
- Controllers logically group similar actions or functionality together
- Actions should contain logic for mapping a request to a business concern

- Example:

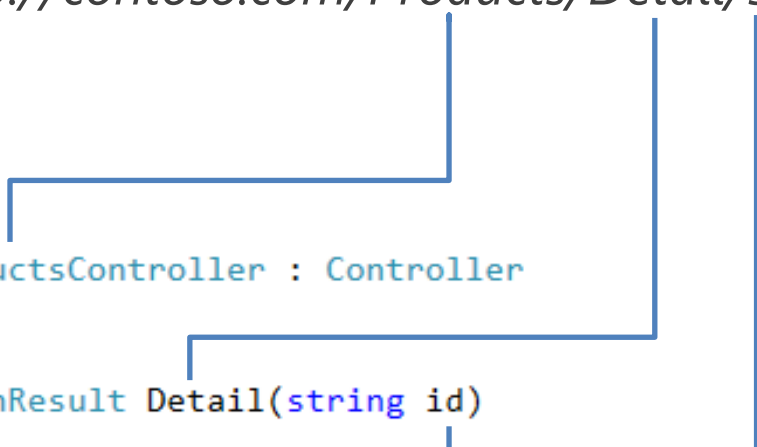


Action Method Parameters

- Parameters on actions are bound to request data and are validated using “Model Binding”
 - Route Data via URL
 - Form Fields
 - Query Strings

• URL: *http://contoso.com/Products/Detail/5*

• Action Method:



```
public class ProductsController : Controller
{
    0 references
    public IActionResult Detail(string id)
    {
        // Retrieve product detail using ID.
        return View();
    }
}
```

Default Routing

- Default Routing Configuration in .NET Core MVC App

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

- Route Name = default
- Pattern: <controller>/<action>/<variable>
- Home & Index are default values
- Id? Is an optional segment which can be used for Parameter Binding.

Demo: Project Organization, Controllers, and Action Methods

Module 4: Controllers

Section 2: Developing Controllers

Lesson: Action Methods Return Types

Action Methods Return Types

- Action methods typically return instance of **IActionResult** or **Task<IActionResult>**
- It is valid to return anything from an action method, though there are some common types.
- Instead of working directly with the Response object, Action Methods return an object that describes what the response should be.
- Action methods are responsible for choosing what kind of response, the action result does the responding.

Common IActionResult Classes - Content

- ViewResult – Renders a view to the response
- PartialViewResult – Renders a partial view to the response
- ViewComponentResult – Renders a view component to the response
- JsonResult – Returns JavaScript Object Notation (JSON) content to the response
- ContentResult – Returns generic “Content” to the response

ViewResult Example

- ViewResult

```
public IActionResult Index()
{
    // Returns /Views/Home/Index.cshtml
    return View();
}
```

- The View() method is inside the Controller base class
- Creates a Microsoft.AspNetCore.Mvc.ViewResult object that renders a view to the response.
- The View() method is smart enough to pick the correct view based on the Controller Name & Action Name.
 - /Views/<controllername>/<actionname>.cshtml

JsonResult Example

- JsonResult

```
public IActionResult GetJsonInfo()  
{  
    return Json(new { SecretMessage = "This is a string message", date = DateTime.Now } );  
}
```

- <http://contoso.com/home/GetJsonInfo>
 - {"secretMessage":"This is a string message","date":"2020-03-16T16:12:31.9953009-05:00"}

ContentResult Example

- ContentResult

```
public IActionResult GetContentResult()  
{  
    return Content("This is a string message");  
}
```

- <http://contoso.com/home/GetContentResult>
 - This is a string message

Common IActionResult Classes – Redirection Results

- These return to the client a redirection response to an action or other destination.
- RedirectResult – Redirect client to the provided URL
- RedirectToRouteResult – Redirects client based on a registered route.
- RedirectToActionResult – Redirects client to a particular action and controller in the same application.

```
public IActionResult PerformRedirect()
{
    return RedirectToAction("Create", "Student");
}
```

Common IActionResult Classes – HTTP Status Code

- These result in an empty response body and return a specific Status Code.
- BadRequestResult – Status Code 400
- ConflictResult – Status Code 409
- NoContentResult – Status Code 204
- NotFoundResult – Status Code 404
- UnauthorizedResult – Status Code 401

```
public IActionResult AccessAdminPage()
{
    return Unauthorized();
}
```

Demo: Action Method Return Types

Module 4: Controllers

Section 2: Developing Controllers

Lesson: Using Parameters and Model Binding

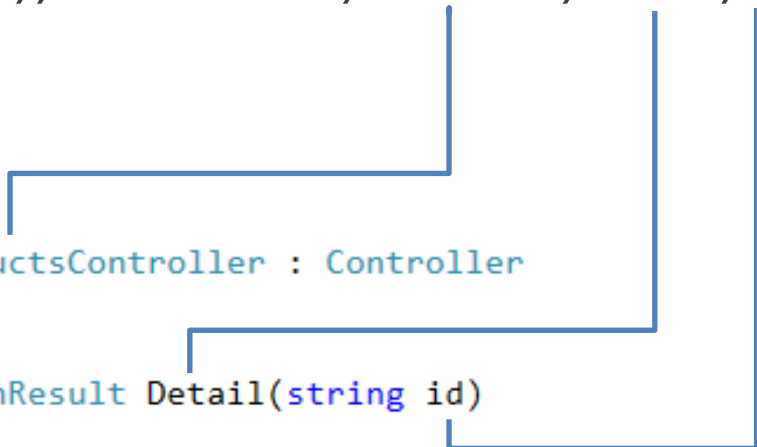
Action Method Parameters

- Parameters on actions are bound to request data and are validated using “Model Binding”
 - Route Data via URL
 - Form Fields
 - Query Strings

• URL: *http://contoso.com/Products/Detail/5*

• Action Method:

```
public class ProductsController : Controller
{
    0 references
    public IActionResult Detail(string id)
    {
        // Retrieve product detail using ID.
        return View();
    }
}
```



The diagram illustrates the binding of the URL parameter '5' to the 'id' parameter of the 'Detail' action method. A blue line starts at the '5' in the URL, goes down, then left, then up to the 'id' parameter in the method signature. Another blue line starts at the 'Detail' method name, goes down, then left, then up to the 'ProductsController' class name, showing the routing path from the URL to the specific action method.

What is Model Binding?

- Controllers and Action Methods work with data that come from HTTP Requests
- For example, we maybe have a StudentID come from the URL, but a Form Field may provide an updated Last Name for our student.
- Writing code to retrieve values from the various locations and convert to the correct .NET Types can be tedious and error prone.
- Model Binding automates this process for us.

What is Model Binding?

- Retrieves data from various sources such as route data, form fields, and query strings
- Provides the data to controllers and Razor pages in method parameters and public properties
- Converts string data to .NET types
- Updates properties of complex types

Simple Model Binding Example

```
public IActionResult GetById(int id, bool dogsOnly)
{
    return Content(String.Format("ID = {0}, Bool = {1}", id, dogsOnly));
}
```

- <https://contoso.com/Home/GetById/2?DogsOnly=true>
- Steps Taken:
 1. Finds the first parameter of method GetById, an integer named id.
 2. Looks through the available sources in the HTTP request and finds id = "2" in route data (URL).
 3. Converts the string "2" into integer 2.
 4. Finds the next parameter of GetById, a boolean named dogsOnly.
 5. Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
 6. Converts the string "true" into boolean true.

No Source for Property

- If no valid value is found for a parameter or property, the property is set to null or default value
- This will be more important when we talk about Complex Types, but for our simple example

```
public IActionResult GetById(int id, bool dogsOnly)
{
    return Content(String.Format("ID = {0}, Bool = {1}", id, dogsOnly));
}
```

- <https://contoso.com/Home/GetById/Five?DogsOnly=true>
- <https://contoso.com/Home/GetById/1?DogsLonely=true>
- <https://contoso.com/Home/GetById/1?DogsOnly=1>

Model Binding Sources

- Model binding gets data in the form of key-value pairs from the following:
 - Form Fields
 - Request Body
 - Route Data / URL
 - Query String Parameters
 - Uploaded Files
- List is scanned in order
- Can use Attributes such as [FromQuery] or [FromForm] specify a specific source.

Model Binding – Simple Types

- Boolean
- Byte, SByte
- Char
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Model Binding – Complex Types

- A Complex Type Example:

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }

    public ICollection<Enrollment> Enrollments { get; set; }
}
```

```
public IActionResult Create(Student studentToCreate)
```

Model Binding - Complex Types

- Complex Types must have:
 - A public default constructor
 - Public writeable properties
- The model binding engines:
 - Instantiates an instance of the class using public default constructor
 - Searches for each property using the Model Binding Sources List
- Sources:
 - Prefix.property_name
 - Prefix = Name of the Parameter, NOT name of the Class
 - E.g. StudentToCreate, NOT Student
 - Property_name

Model Binding – Complex Types Customizations

- There are many attributes which can be applied to customize binding for Complex Types
- `[Bind(Prefix = "Estudiante")] Student studentToCreate`
- `[BindNever]`
- `[BindRequired]`
- `[Bind("LastName", "FirstMidName", "EnrollmentDate")]`
 - Can be applied to Class or inside Method
 - Can be used to protect against **overposting**.

Model Binding – Model State

- Model state represents errors from:
 - Model Binding
 - Model Validation
- Model Binding & Model Validation occur before execution of Action Method
- We can use ModelState.IsValid to check the results.
- Can rerun validation using TryValidateModel

Demo: Model Binding

Module 4: Controllers

Section 2: Developing Controllers

Lesson: Action Filters and other Attributes

Action Filters

- Classes which apply Cross-Cutting Concerns to Controllers and Action Methods.
- Examples
 - Authentication/Authorization
 - Caching
 - Exception Handling
- Can be applied using Attributes to Controllers & Action Methods
- Can also apply a global filter by adding into `Startup.ConfigureServices`

Filter Attributes

Filter Attribute	Description
[Authorize]	To restrict access to only those users that are authenticated and authorized
[Authorize(Roles="Admin")]	To restrict access only to users with Role "Admin"
[AllowAnonymous]	To allow anonymous users to call the action method
[RequireHttps]	To force an unsecured HTTP request to be resent over HTTPS
[ValidateAntiForgeryToken]	To prevent forgery of a request (Defense against CSRF attack)

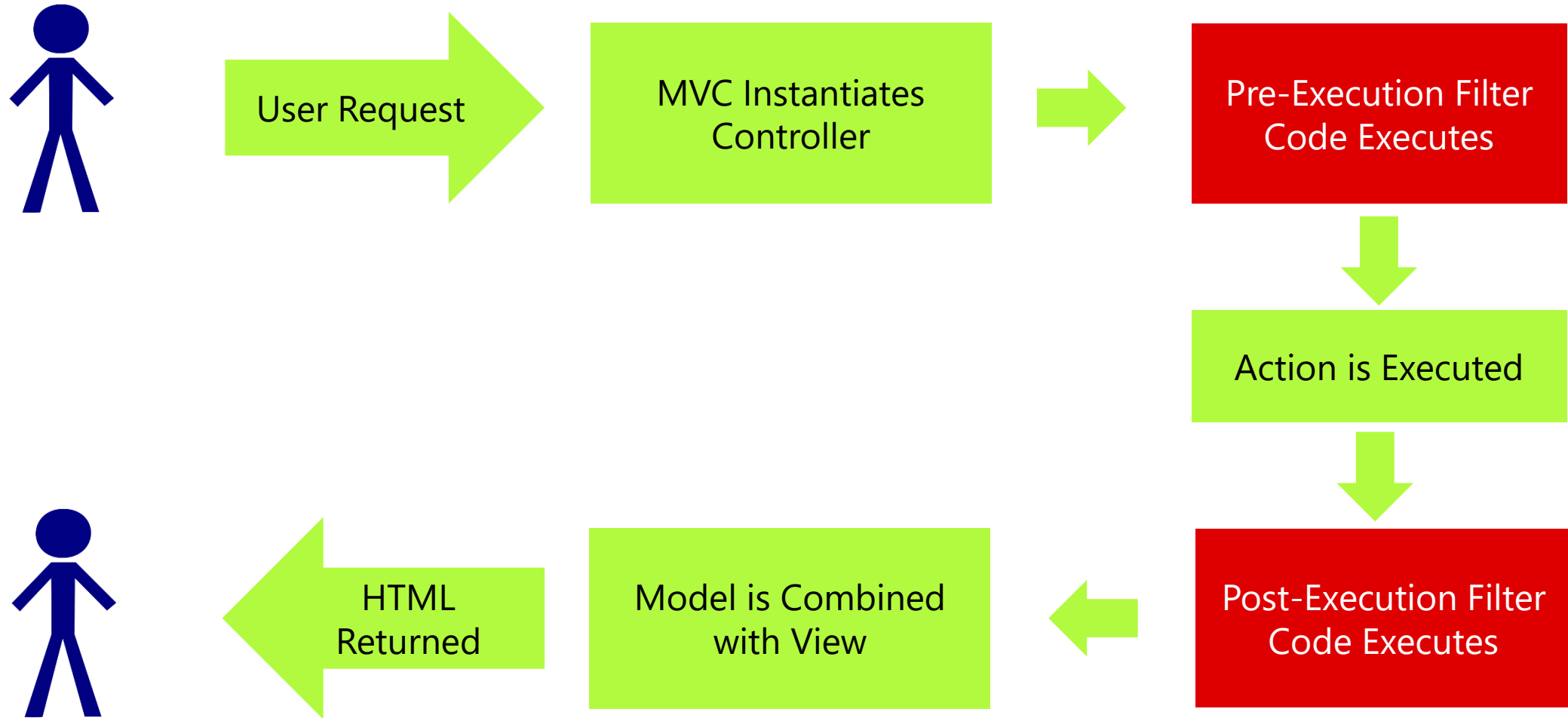
Action Filter Example

- Below are Two Action Filters: AllowAnonymous and ValidateAntiForgeryToken
- HttpPost is a “Selector Attribute” and we will discuss it shortly.

```
//  
// POST: /Account/Register  
[HttpPost]  
[AllowAnonymous]  
[ValidateAntiForgeryToken]  
0 references  
public async Task<IActionResult> Register(RegisterViewModel model)  
{
```

Account Controller - Register Method

Actions with Filters



Custom Filters

- Its relatively easy to create your own filters

```
public class MySampleActionFilter : Attribute, IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        Debug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        Debug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

```
[MySampleActionFilter]
public IActionResult Index()
{
    // Returns /Views/Home/Index.cshtml
    return View();
}
```

Selector Attributes

Selector Attribute	Description
[NonAction]	To mark a method as non-action
[HttpPost]	To restrict a method to handling only HTTP POST requests
[HttpPut]	To restrict a method to handling only HTTP PUT requests
[HttpGet]	To restrict a method to handling only HTTP GET requests
[AcceptVerbs]	Specifies which HTTP verbs an action method will respond to
...	...

Demo: Filters

Module 4: Controllers

Section 3: Dependency Injection (DI)

Lesson: Overview

Dependency Injection

- Dependency Injection is a technique for achieving Inversion of Control between classes and their dependencies
- Dependency: Any object that another object requires
 - Example: StudentController may depend on a data store for loading/saving Students.

```
public class StudentController : Controller
{
    SomeStudentDB _db = new SomeStudentDB();

    public IActionResult Index()
    {
        return View(_db.GetStudentList());
    }
}
```

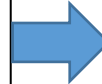
Dependency Injection – Issues with example

- We must modify our StudentController if we want to use a different implementation
- If our dependency has its own dependencies, they must be configured by StudentController.
 - This can have a large cascading effect
 - Imagine a logging component that logs to a database
- This model is very difficult to unit test.
 - We may need to use mock or stub classes to replace our dependency

Dependency Injection

```
public class StudentController : Controller
{
    SomeStudentDB _db = new SomeStudentDB();

    public IActionResult Index()
    {
        return View(_db.GetStudentList());
    }
}
```



```
public class StudentController : Controller
{
    ISomeStudentDB _db;

    public StudentController(ISomeStudentDB db)
    {
        _db = db;
    }

    public IActionResult Index()
    {
        return View(_db.GetStudentList());
    }
}
```

- Use of Interface to abstract dependency
- **Injection** of the dependency into controller's constructor.

Dependency Injection – Registering Dependencies

- Dependencies are registered during application startup in Startup.cs method named **ConfigureServices**

```
// This method gets called by the runtime. Use this method to add services to the container.  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddSingleton<ISomeStudentDB, SomeStudentDB>();  
    services.AddControllersWithViews();  
}
```

Framework – Provided DI in ASP.NET Core

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      // Add framework services.
4      services.AddEntityFramework()
5          .AddSqlServer()
6          .AddDbContext<ApplicationDbContext>(options =>
7              options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));
8
9      services.AddIdentity<ApplicationUser, IdentityRole>()
10         .AddEntityFrameworkStores<ApplicationDbContext>()
11         .AddDefaultTokenProviders();
12
13     services.AddMvc();
14
15     // Add application services.
16     services.AddTransient<IEmailSender, AuthMessageSender>();
17     services.AddTransient<ISmsSender, AuthMessageSender>();
18 }
```

DI is provided by ASP.NET Core, and is used to resolve Controllers as well as services

Framework services are set up in the ConfigureServices function in ASP.NET Core

Your own services can also be registered here – specifying Interface, Class, and Lifetime

Service Lifetimes and Registration Options

Transient

- Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless service

Scoped

- Scoped lifetime services are created once per request

Singleton

- Singleton lifetime services are created the first time they are requested, and then every subsequent request will use the same instance. If your application requires singleton behaviour, allowing the services container to manage the service's lifetime is recommended instead of implementing the singleton design pattern and managing your object's lifetime in the class yourself

Instance

- Similar to Singleton, but the instance is created when added to the services container and reused throughout, whereas with a Singleton the instance is first created only when first requested for use

Other Dependency Injection

- Services can be registered against Interface, Base Class or Concrete Class
- Services can be injected directly into an Action Method using [FromServices]
- Views are able to have services injected to them using @inject
 - Think of @inject as adding a property to the view
- Services.TryAdd{Lifetime} e.g. Services.TryAddSingleton< ... > will only register a dependency if there is not already a registered implementation.
- Dependency Injection allows us to focus on developing Controllers & Views without concern for constructing dependencies.

Demo: Dependency Injection

Module 4: Controllers

Section 5: Summary

Lesson: Best Practices &
Summary

Controller Best Practices

- Keep controllers lightweight
 - Use filters
- High Cohesion
 - Make sure all actions are closely related
- Low Coupling
 - Controllers should know as little about the rest of the system as possible
 - Simplifies testing and changes

Module Summary

- Controller and its role in MVC pattern
- Controller development
- Action methods
- Action filters
- Dependency Injection
- Controller best practices



Lab: Controllers



