

.NET Framework: Developing Modern Web Apps with ASP.NET MVC – Workshop*PLUS*

Wael Kdounh

Senior Consultant

v1.0

Module 3: Controllers

Module Overview

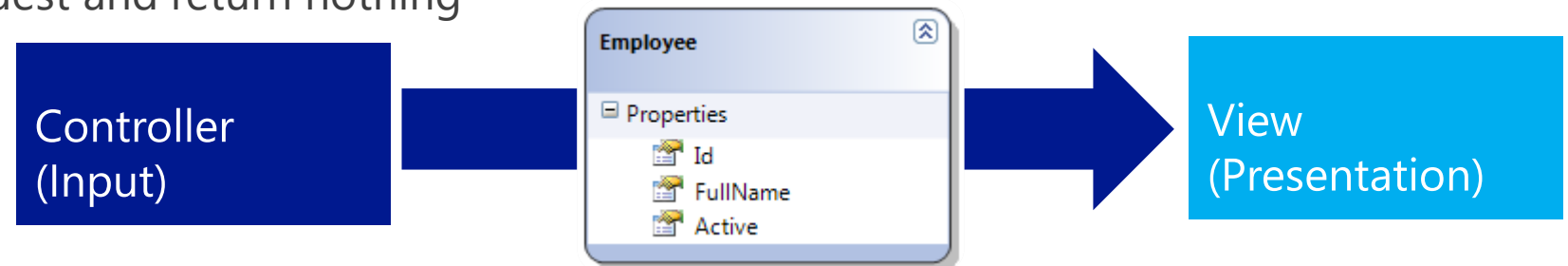
Module 3: Controllers

Section 1: Controller Fundamentals

Lesson: Role of Controllers

Controller

- Controller is responsible for:
 - Responding to requests made against ASP.NET MVC
 - Manipulating the model (if necessary)
 - Selecting a View to send back to user via response
- Each browser request is mapped against a particular controller
- Controller may:
 - Return a view
 - Redirect the user to another controller
 - Perform action on the request and return nothing



Role of Controller

- Controller handles and responds to user input and interaction
- 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 3: Controllers

Section 2: Developing Controllers

Lesson: Controller Development

Controller Development

- Controller class inherits from `System.Web.Mvc.Controller`
- Controller class name has 'Controller' suffix so that the routing engine can find it
 - For example: `HomeController`, `AccountController`, etc.

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

    public ActionResult About()
    {
        ViewBag.Message = "Your application description page.";

        return View();
    }
}
```

Default Controllers

- HomeController
 - Responsible for the home page at the root of the website
- AccountController (optional, depends on authentication model chosen)
 - Responsible for application security, such as login and registration
- ManageController (optional, depends on authentication model chosen)
 - Responsible for account management tasks, such as password change

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe)
```


Default Model Binder

- Default Model Binder automatically populates controller action parameters
 - Takes care of mundane property mapping and type conversion
 - Uses the name attribute of input elements
 - Automatically matches parameter names for simple data types
 - Complex objects are mapped by property name; use dotted notation

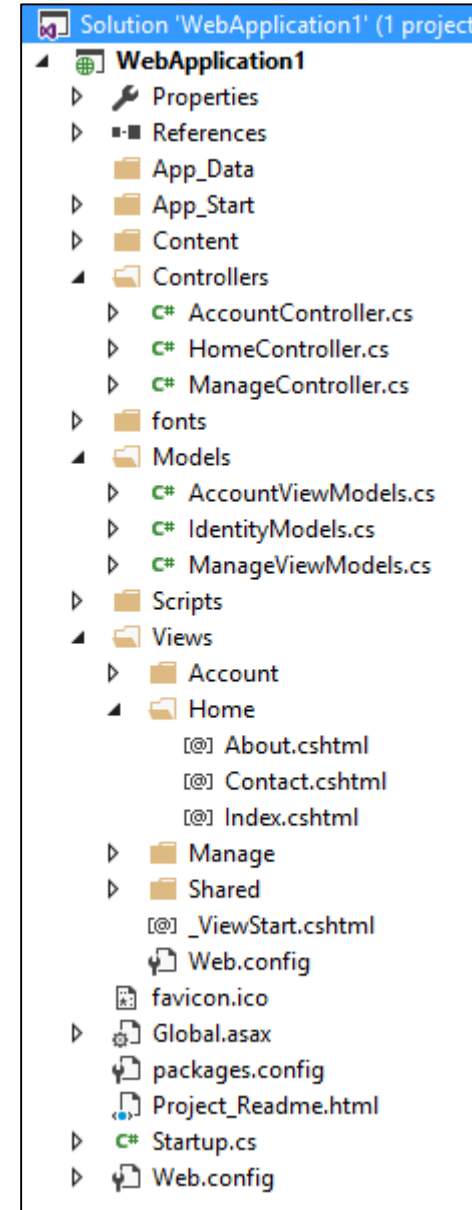
```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
    }
}
```

```
public class RegisterViewModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

The diagram illustrates the mapping of properties from the RegisterViewModel class to the Register action parameters. Arrows show the following connections: from 'Email' to 'model.Email', from 'Password' to 'model.Password', and from 'ConfirmPassword' to 'model.Password'.

ASP.NET MVC Project File Organization

- Default conventions can be overridden
 - ASP.NET 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 MVC looks for the View template file in **`\Views\[ControllerName]\`**



Demo: Organization of Controllers, Views, and Models

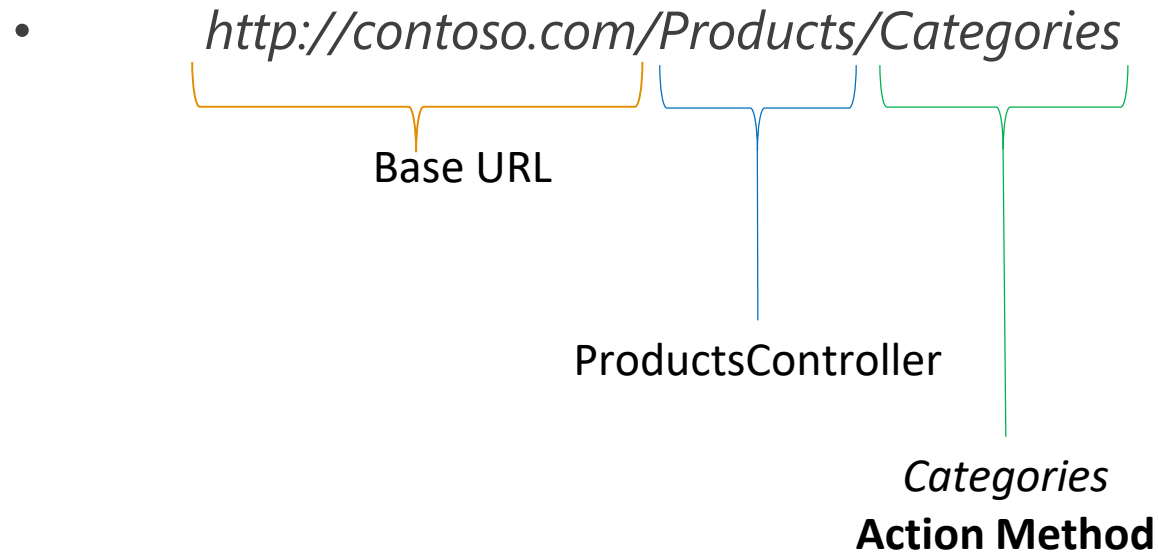
Module 3: Controllers

Section 2: Developing Controllers

Lesson: Action Methods

Action Methods

- Action methods have 1:1 mapping with user interaction
 - Form submission, clicking a link, etc.
- User actions lead to action method calls
- Task-based asynchronous methods supported
- Example:



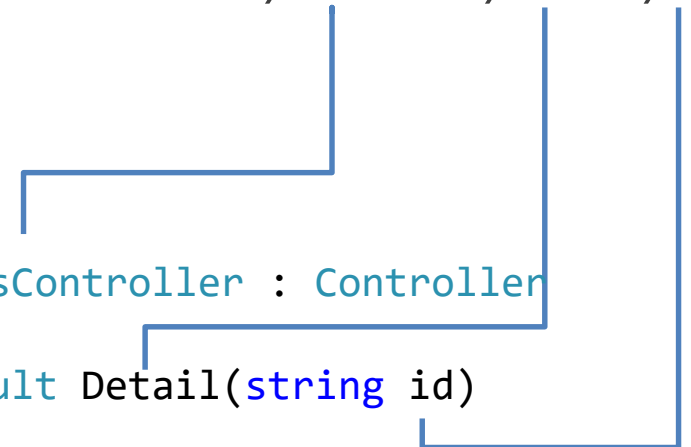
Action Method Parameters

- Query string or form post parameters are passed into Action methods as parameters

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

- Action Method:

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

A diagram consisting of blue lines that maps parts of the URL 'http://contoso.com/Products/Detail/5' to the corresponding parts of the C# code. A line connects 'http://contoso.com/' to the class 'ProductsController'. Another line connects '/Products/' to the class 'ProductsController'. A third line connects '/Detail/' to the method 'Detail'. Finally, a line connects the number '5' to the parameter 'id' in the method signature.

Action Method Results

- The Action method typically returns a View, but not always

```
public class StoreController : Controller
{
    // GET: /Store/
    public ActionResult Index()
    {
        return View();
    }

    // GET: /Store/Details
    public ActionResult Details()
    {
        return View("Details");
    }

    // GET: /Store/Browse?genre=Disco
    public string Browse(string id)
    {
        return "Store.Browse, Genre = " + id;
    }
}
```

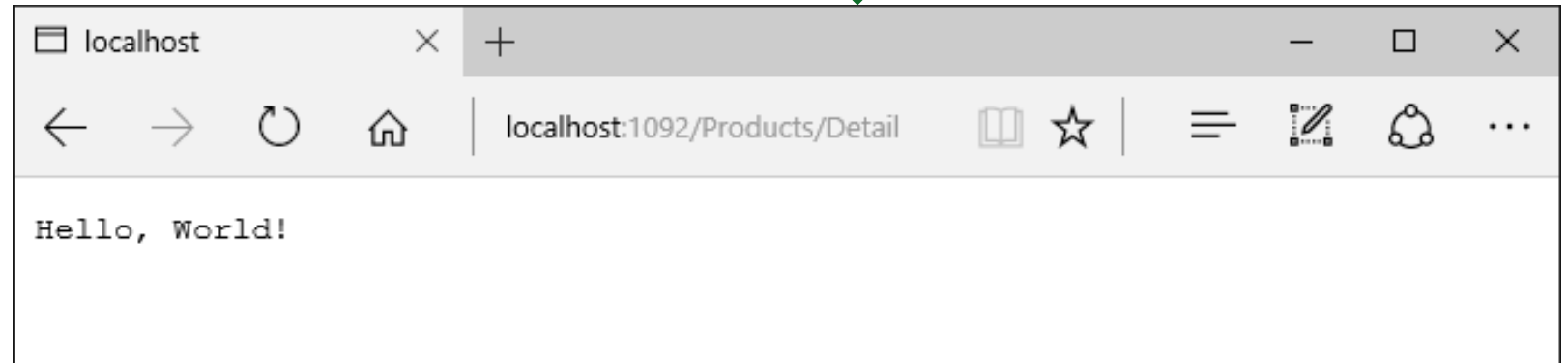
```
public FileResult SomeFile()
{
    string filename = @"C:\Example.pdf";
    string contentType = "application/pdf";
    string downloadName = "ExampleFile.pdf";

    return File(filename, contentType, downloadName);
}
```

Built-in Action Return Types

- All derive from **ActionResult**
 - ViewResult
 - PartialViewResult
 - RedirectToRouteResult
 - RedirectResult
 - ContentResult
 - FileResult
 - JsonResult
 - JavaScriptResult
(use with care)
 - UnauthorizedResult
 - HttpNotFoundResult
 - HttpStatusCodeResult
 - EmptyResult

```
public ContentResult Index()  
{  
    return Content("Hello, World!");  
}
```



Demo: Controller Development

Module 3: Controllers

Section 2: Developing Controllers

Lesson: Filters

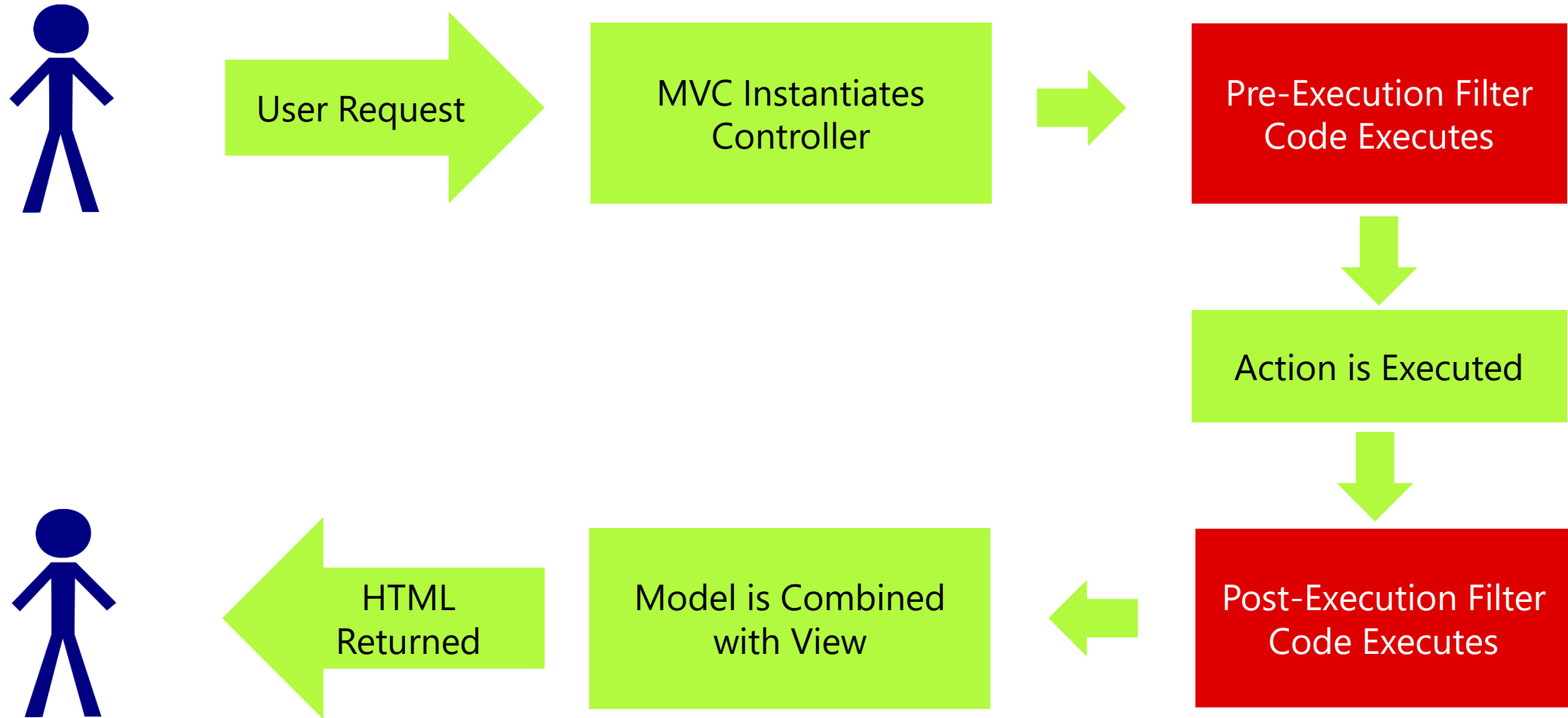
Action Filter and Selector Attributes

- Public controller methods are actions by default
- **Selectors** differentiate Actions
- **Filters** control access

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

Account Controller - Register Method

Actions with Filters



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
...	...

Filter Attributes

Selector Attribute	Description
[AllowAnonymous]	To allow anonymous users to call the action method
[Authorize]	To restrict access to only those users that are authenticated and authorized
[RequireHttps]	To force an unsecured HTTP request to be resent over HTTPS
[ValidateAntiForgeryToken]	To prevent forgery of a request (Defense against CSRF attack)
[ValidateInput]	To mark action methods whose input must be validated
[HandleError]	To handle an exception that is thrown by an action method
...	...

Demo: Filters

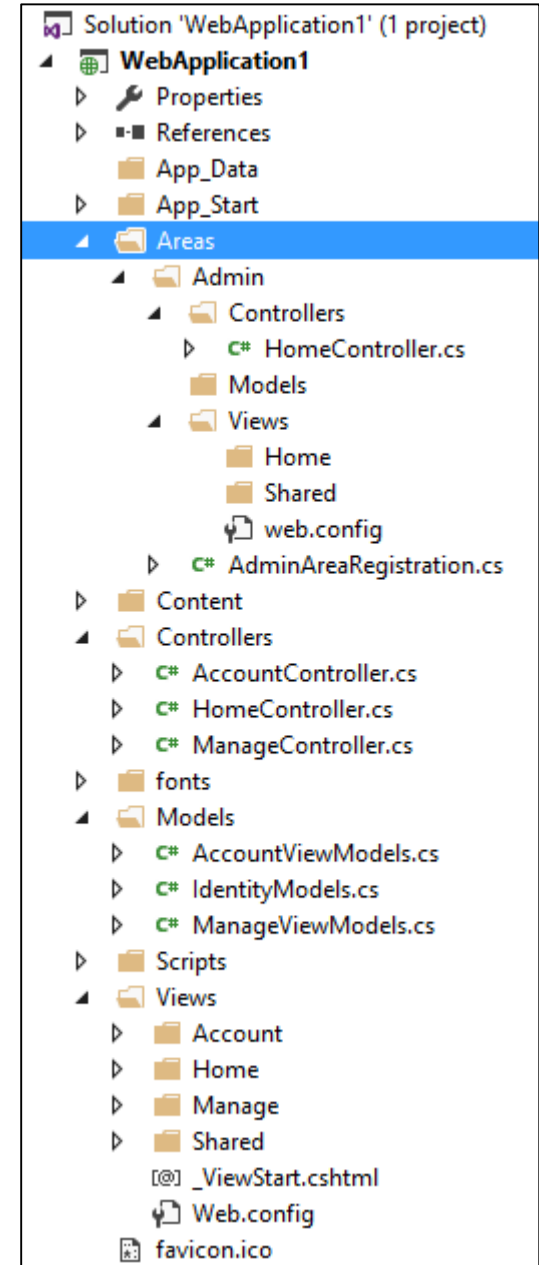
Module 3: Controllers

Section 2: Developing Controllers

Lesson: Areas

Areas

- MVC Areas separate a large MVC application into smaller functional groupings
 - For example: large e-commerce site divided into areas for storefront, product reviews, user accounts, etc.
- Area Guidelines
 - Areas directory must exist as a project child directory
 - Areas contains subdirectory for each area
 - Controllers should be located at:
/Areas/[area]/Controllers/[controller].cs
 - Views should be located at:
/Areas/[area]/Views/[controller]/[action].cshtml
 - Controller names in different areas may overlap



Area Registration

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void
    {
        AreaRegistr
        FilterConfi
        RouteConfig
        BundleConfi
    }
}
```

```
public class AdminAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get
        {
            return "Admin";
        }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute(
            "Admin_default",
            "Admin/{controller}/{action}/{id}",
            new { action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

No default controller
(can be added though)

Same API as
used in route
setup

Routes begin
with a prefix

Areas\Admin\AdminAreaRegistration.cs

Area Linking

_Layout.cshtml

```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
  </ul>
  @Html.Partial("_LoginPartial")
</div>
```

Area becomes a
named routing
parameter

Null is just there
to use the right
overload

Change it to:

```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home", new { area = "" }, null)</li>
    <li>@Html.ActionLink("Admin", "Index", "Home", new { area = "Admin" }, null)</li>
    <li>@Html.ActionLink("About", "About", "Home", new { area = "" }, null)</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home", new { area = "" }, null)</li>
  </ul>
  @Html.Partial("_LoginPartial")
</div>
```

Links to non-
area scoped
actions require
empty string

Demo: Areas

Module 3: Controllers

Section 3: Advanced Controller Design

Lesson: Advanced Controller Design

Asynchronous Controller Action Methods

- ASP.NET MVC supports asynchronous action methods

```
public class PortalController : Controller {
    public async Task<ActionResult> Index(string city)
    {
        NewsService newsService = new NewsService();
        WeatherService weatherService = new WeatherService();
        SportsService sportsService = new SportsService();

        var newsTask = newsService.GetNewsAsync(city);
        var weatherTask = weatherService.GetWeatherAsync(city);
        var sportsTask = sportsService.GetScoresAsync(city);

        await Task.WhenAll(newsTask, weatherTask, sportsTask);

        PortalViewModel model = new PortalViewModel
        {
            newsTask.Result,
            weatherTask.Result,
            sportsTask.Result
        };
        return View(model);
    }
}
```

Sync vs. Async Action Methods

- Use **synchronous** methods when:
 - The operations are simple or short-running
 - Simplicity is more important than efficiency
 - The operations are primarily CPU operations instead of operations that involve extensive disk or network overhead
- Use **asynchronous** methods when:
 - You are calling services
 - The operations are network-bound or I/O-bound instead of CPU-bound
 - Parallelism is more important than simplicity of code
 - Enabling cancelation of a long-running request

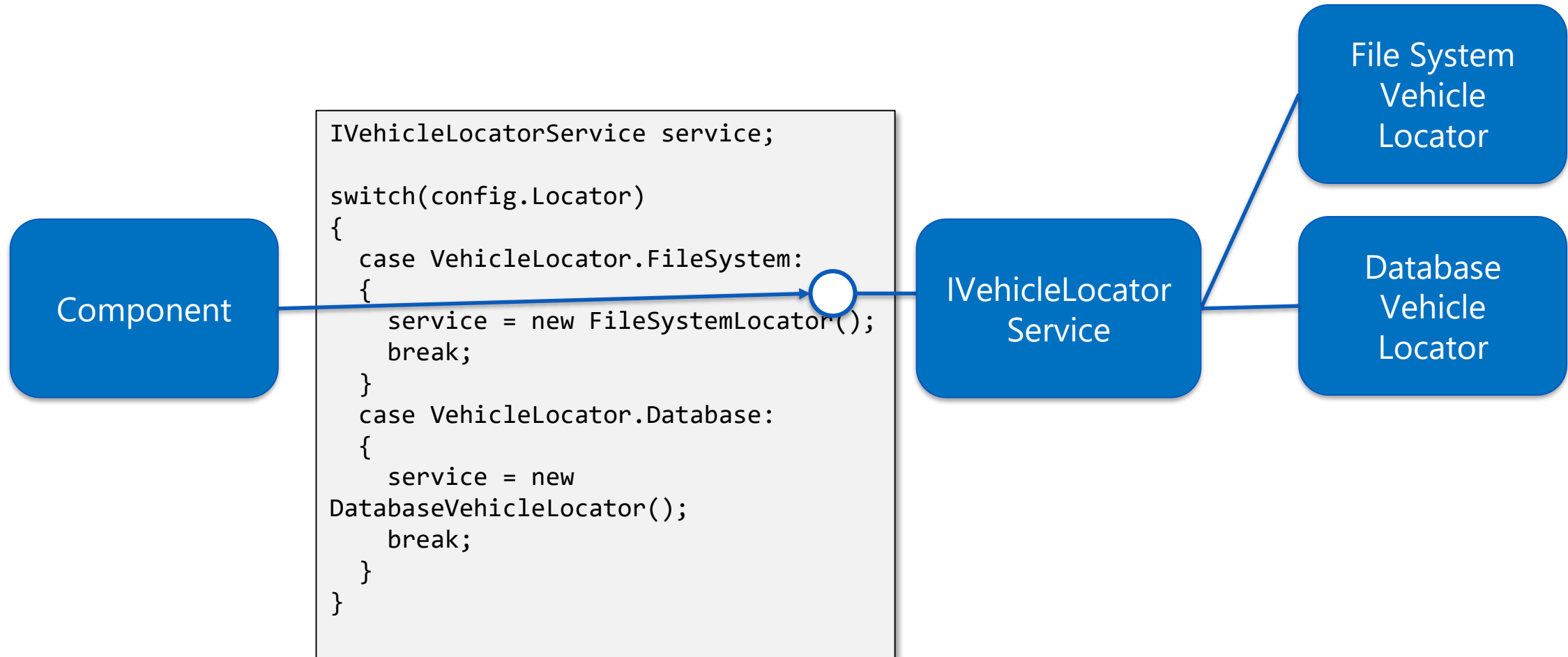
Module 3: Controllers

Section 4: Dependency Injection (DI)

Lesson: Overview

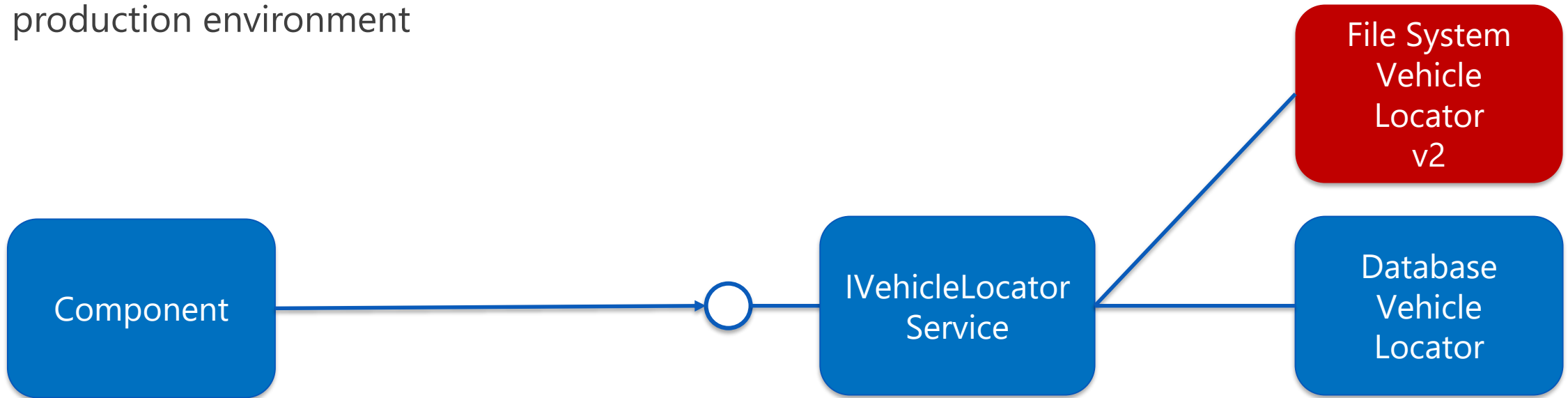
Without Dependency Injection (DI)

Tightly Coupled - Component determines which implementation to use



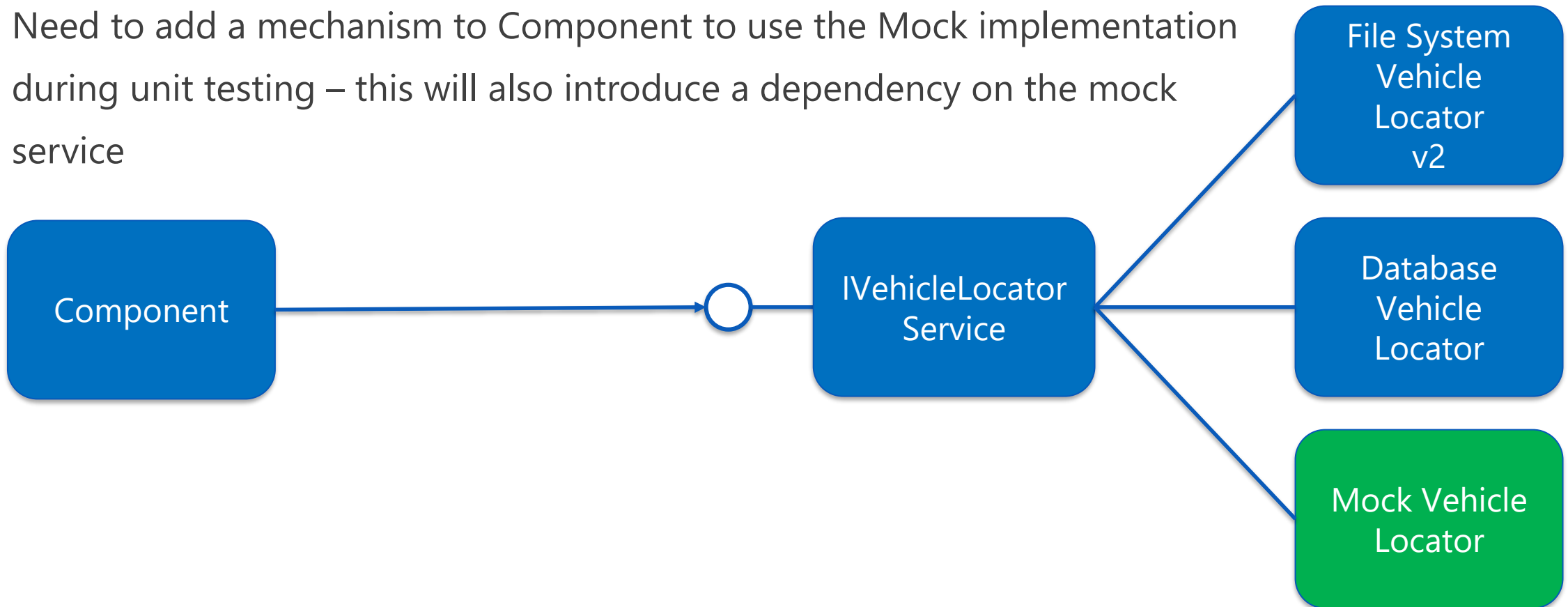
What's the Problem? (Part 1)

- Redeploy of new service that is tightly coupled means Component also needs to be rebuilt, retested, and redeployed
- Both components will be shipped/deployed, whereas only one may be required for the production environment



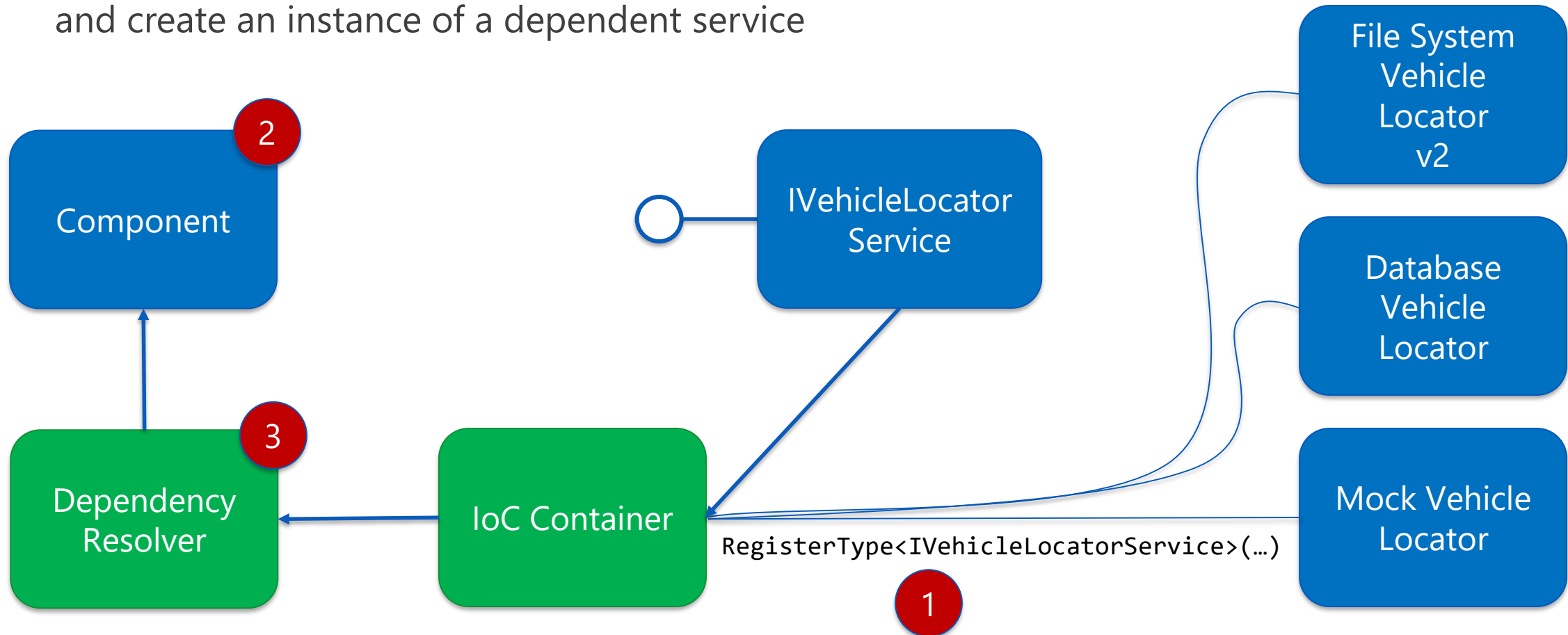
What's the Problem? (Part 2)

- Unit testing should be proving Component in isolation, and not rely on external resources such as databases or file systems
- Need to add a mechanism to Component to use the Mock implementation during unit testing – this will also introduce a dependency on the mock service



Inversion of Control

- DI is a design pattern which removes the responsibility of a component to know how to locate and create an instance of a dependent service



IoC Containers for .NET

Comparison based on the book “Dependency Injection in .NET” by Mark Seemann:

DI Container	Advantages	Disadvantages	Project link
Castle Windsor	Complete Understands Decorator Typed factories Commercial support available	Quirky API in places	https://github.com/castleproject/Windsor
StructureMap	Just works in many cases	No interception	https://github.com/structuremap/structuremap
Spring.NET	Interception Comprehensive documentation Commercial support available	Very XML-centric No convention-based API No custom lifetimes Limited auto-wiring	https://github.com/spring-projects/spring-net
Autofac	Easy to learn API Commercial support available	No interception Partial support for custom lifetimes	https://github.com/autofac/Autofac
Unity	Interception Good documentation Consistent API	Lifetime management pitfalls	https://github.com/unitycontainer/unity
Ninject	Interception (as extension) Simple API	Documentation lags behind releases	https://github.com/ninject/Ninject

DI in ASP.NET MVC

- Supported via **IDependencyResolver** interface:

```
public interface IDependencyResolver
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

- Implement this interface in a class and pass it on during app startup:

```
DependencyResolver.SetResolver(new MyDependencyResolver());
```

- MVC will use this resolver to create your controllers
- Your containers in turn should rely on services supplied via interface parameters in the constructor
 - Less optimal: properties storing these dependencies

Implementing IDependencyResolver

- Typically, you won't implement your own resolver
 - Creating a good resolver is a complex task
- Instead, you delegate type resolution requests to an Inversion-Of-Control (or IoC) Container
 - Unity, Ninject, Castle.Windsor, Autofac, StructureMap, etc.
- The most straightforward solution is to use Microsoft's Unity
 - Unity is a flexible and configurable IoC container
 - Its MVC bootstrapper (<https://www.nuget.org/packages/Unity.Mvc>) modifies your project to use Unity with minimal work, including code templates
- In your startup code or **UnityConfig.RegisterTypes** for Unity.Mvc, register your types with Unity:
`container.RegisterType<IProductRepository, ProductRepository>();`
- Alternatively, use **container.LoadConfiguration();** to setup Unity from web.config

Service Lifetimes and Registration Options in Unity

TransientLifetimeManager – default

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

PerRequestLifetimeManager

- Scoped lifetime services are created once per HTTP request.

ContainerControlledLifetimeManager with RegisterType() or default for RegisterInstance()

- Instance created the first time it's requested (RegisterType() only), every subsequent request will use the same instance. If your application requires singleton behaviour, this is recommended instead of implementing the singleton design (anti-) pattern and managing your object's lifetime in the class yourself. Also: the container will **dispose** your object at the end of its lifetime.

ExternallyControlledLifetimeManager with RegisterInstance()

- Similar to ContainerControlledLifetimeManager, but the instance is created when added to the services container and reused throughout, whereas ContainerControlledLifetimeManager with RegisterType() creates the instance only when first requested for use. Container will never dispose the instance.

Types of DI in Unity

- Constructor Injection
- Property (Setter) Injection

```
public class CalcController : Controller
{
    // Inject via property
    [Dependency]
    public ISmsSender SmsSender { get; set; }
    private readonly ILogger _logger;

    // Inject via constructor
    public CalcController(ILogger logger)
    {
        _logger = logger;
    }
}
```

Demo: Dependency Injection

Module 3: Controllers

Section 5: Caching

Lesson: Overview

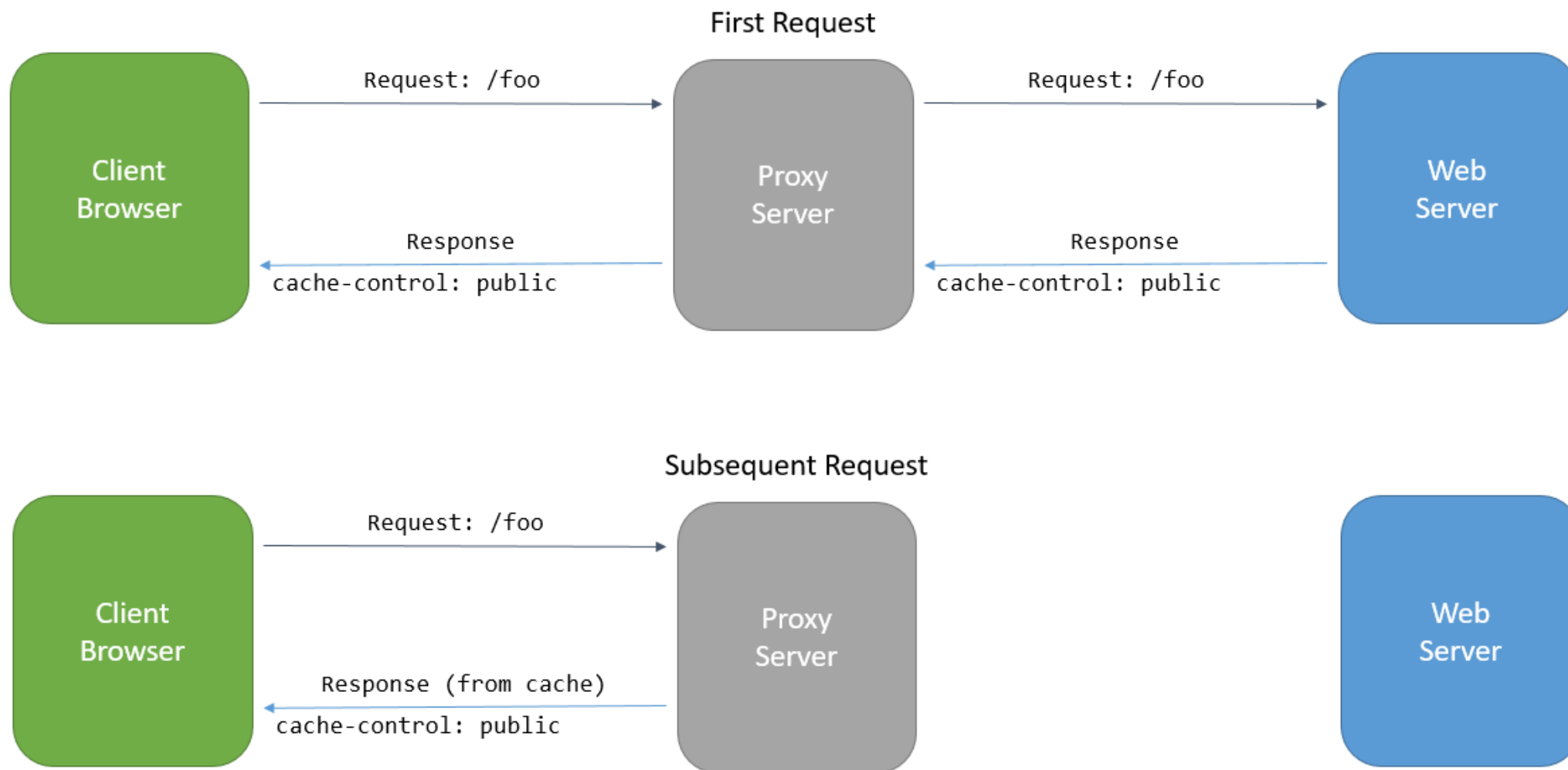
Caching

- In-memory Caching
 - Stored in memory of a single server
- Distributed Caching
 - Shared by multiple servers
 - Used by server-farm hosted apps
- Response Caching
 - Controls browser's client cache
- Output Caching
 - Caches controller action responses

In-Memory Caching

- Do not use `HttpContext.Current.Cache`
 - Makes unit testing harder
 - Can't be used in async methods (may run on different thread)
- Better: use DI to supply the cache for your controllers
- Use the new **`System.Runtime.Caching.MemoryCache`** class as implementation
 - It offers thread-safety and cache expiration policies
 - Allows multiple cache instances
 - Does not depend on `System.Web.dll`

Response Caching



OutputCacheAttribute

- Implements both response and output caching in ASP.NET MVC
- Fine-grained control on location (any, client, server, downstream, etc.)
- Allows caching of multiple (likely different) results based on:
 - Content-Encoding
 - HTTP Header
 - Action parameter

```
[OutputCache(VaryByParam = "name")]  
public ActionResult Greetings(string name)  
{  
    return View(name);  
}
```

Demo: In-Memory Caching

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
 - Repository pattern
 - Wrap data context calls into another object

Module Summary

- Controller and its role in MVC pattern
- Controller development
- Action methods
- Action filters
- Asynchronous controller actions
- Error-Logging Module
- Dependency Injection
- Caching
- Controller best practices



Lab: Controllers



