

Developing Cross-Platform Web Apps With Blazor

Wael Kdouh - @waelkdouh

Senior Customer Engineer

v1.0

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

© 2013 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/about/legal/permissions/>

Active Directory, Azure, IntelliSense, Internet Explorer, Microsoft, Microsoft Corporate Logo, Silverlight, SharePoint, SQL Server, Visual Basic, Visual Studio, Windows, Windows Server, and Windows Vista 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.

How to View This Presentation

- To switch to **Notes Page** view:
 - On the ribbon, click the **View** tab, and then click **Notes Page**
- To navigate through notes, use the Page Up and Page Down keys
 - Zoom in or zoom out, if required
- In the **Notes Page** view, you can:
 - Read any supporting text—now or after the delivery
 - Add notes to your copy of the presentation, if required
- Take the presentation files home with you

Module 6: Services and Dependency Injection

Module Overview

Module 6: Services and Dependency Injection

Section 1: Services and Dependency Injection

Lesson: Overview

What Is Dependency Inversion?

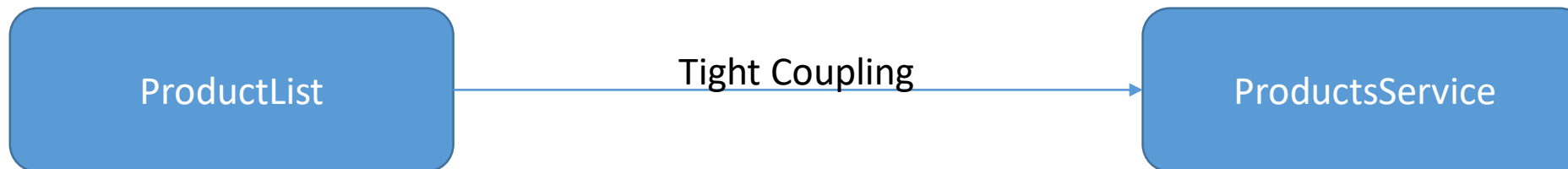
- Data is typically stored in a database on a server. Retrieving and storing this data can be done in the component itself, but this is a bad idea
- Instead you should put this logic into a service object. A service object's role is to encapsulate specific business rules, especially how data is communicated between the client and server. A service object is also a lot easier to test since you can write unit tests that run on their own, without requiring a user to interact with the application for testing

Understanding Dependency Inversion

- Imagine a component that uses a service. The component creates the service using the new operator:

```
@using MyFirstBlazor.Client.Services
<div>
    @foreach (var product in productService.GetAllProducts())
    {
        <div>@product.Name</div>
        <div>@product.Description</div>
        <div>@product.UnitPrice</div>
    }
</div>
@Code {
    ProductService productService = new ProductService();
}
```

We just created a dependency between the ProductList component and the ProductService



Problems Resulting From Tight Coupling

- Testing ProductList component requires a server on the network to talk to. If the server is not ready yet, you cannot test your component
- Replacing ProductsService becomes challenging as now you need to find every use of the ProductsService in your application and replace the class

Using Dependency Inversion Principle

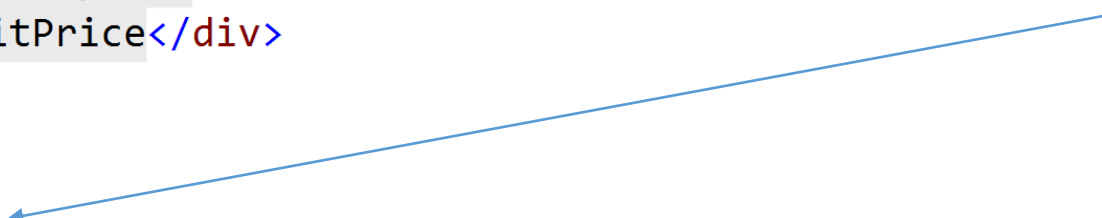
- The Dependency Inversion Principle states that:
 - High-level modules should not depend on low-level modules. Both should depend on abstractions
 - Abstractions should not depend on details. Details should depend on abstractions
- This means the ProductList component (the higher-level module) should not directly depend on ProductService (the lower-level module)

Using Dependency Inversion Principle

- ProductList component should rely on an abstraction. It should rely on an interface describing what a ProductsService should be able to do, not a class describing how it should work

```
<div>
  @foreach (var product in productService.GetAllProducts())
  {
    <div>@product.Name</div>
    <div>@product.Description</div>
    <div>@product.UnitPrice</div>
  }
</div>
@code {
  IProductsService productService;
}
```

ProductList component
now relies on abstraction



Using Dependency Inversion Principle

- Now the ProductList component only relies on the IProductsService interface, an abstraction

Make the ProductsService implement the interface

```
public class ProductsService : IProductsService
{
    3 references | 0 exceptions
    public List<Product> GetAllProducts()
    {
        // implementation goes here...
        return new List<Product>();
    }
}
```

If you want to test the ProductList component with dependency inversion in place, you can simply build a hard-coded version

```
public class HardCodedProductsService : IProductsService
{
    public static List<Product> products = new List<Product>
    {
        new Product {
            Id =1,
            Name = "My Custom Product",
            Description = "Best product!",
            UnitPrice = 40
        },
        new Product {
            Id =2,
            Name = "My Second Custom Product",
            Description = "Second Best product!",
            UnitPrice = 30
        },
    };

    3 references | 0 exceptions
    public List<Product> GetAllProducts()
    {
        return products;
    }
}
```

Using Dependency Inversion Principle

- By applying the Principle of Dependency Inversion, you gained a lot more flexibility



Module 6: Services and
Dependency Injection view

Section 1: Services and
Dependency Injection

Lesson: Adding Dependency
Injection

Adding Dependency Injection

- If you try to run the ProductList Component you will get a NullReferenceException

The screenshot shows a web browser displaying a Blazor application. The application has a dark theme and a sidebar on the left with the following menu items: Home, Products List (selected), Counter, and Fetch data. The main content area displays "Hello, world!" and "Welcome to your new app." Below this is a survey prompt: "How is Blazor working for you? Please take our [brief survey](#) and tell us what you think."

At the bottom of the browser window, the developer console is open, showing several error messages. The first two are related to failed resource loads (404 status). The third error, which is highlighted with a red box, is a Wasm error: "WASM: Unhandled exception rendering component: System.NullReferenceException: Object reference not set to an instance of an object." The fourth error is a Wasm error related to the rendering of the ProductList component, also highlighted with a red box. The console messages are as follows:

- Debugging hotkey: Shift+Alt+D (when application has focus)
- Failed to load resource: the server responded with a status of 404 ()
- WASM: Unhandled exception rendering component: System.NullReferenceException: Object reference not set to an instance of an object.
- WASM: at Module5.Pages.ProductList.BuildRenderTree (Microsoft.AspNetCore.Components.Rendering.RenderTreeBuilder __builder) [0x0001c] in C:\Users\waekdo\OneDrive - Microsoft\Consulting\WorkShops\Blazor\Demos\Module5\Module5\Pages\ProductList.razor:3

Adding Dependency Injection

- ProductsList component requires an instance of a class implementing IProductsService

```
@page "/productlist"
<div>
    @foreach (var product in productService.GetAllProducts())
    {
        <div>@product.Name</div>
        <div>@product.Description</div>
        <div>@product.UnitPrice</div>
    }
</div>
@code {
    // TODO: This will throw a NullReferenceException if not instantiated
    IProductsService productService;
}
```

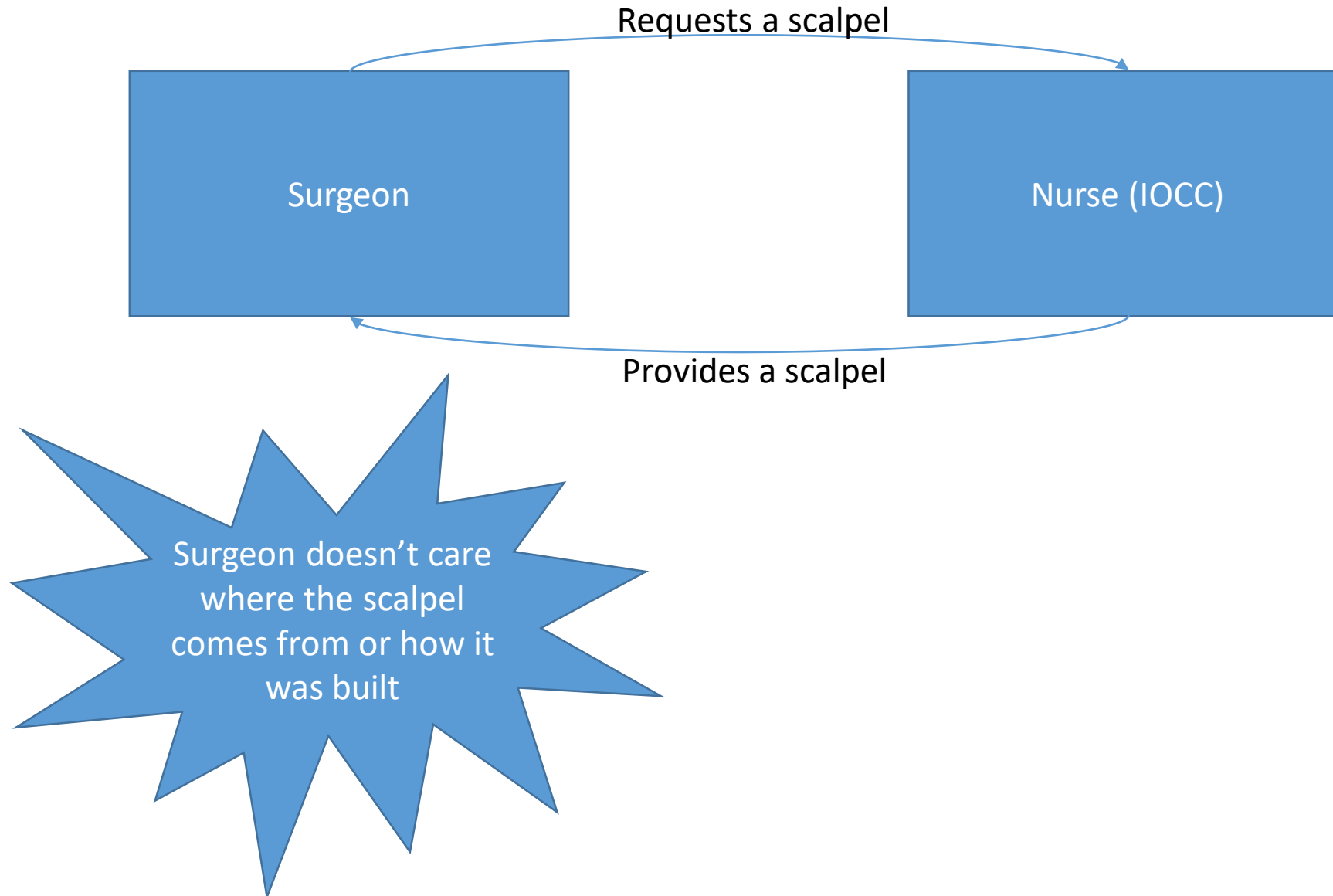
Adding Dependency Injection

- We can add Dependency Injection by passing the ProductService in the constructor of the ProductList component.
 - For example `new ProductList(new ProductService())`
- But if the ProductService also depends on another class, it creates a Deep Chain of Dependencies Manually: `new ProductList(new ProductService(new Dependency()))`
- This is, of course, not a practical way

Adding Dependency Injection

- A **better approach** is to use an **Inversion-of-Control Container** (IOCC)
- An Inversion-of-Control Container (IoCC) is just another object that specializes in creating objects for you
- You simply ask it to create an instance of a class and it will take care of creating any dependencies required

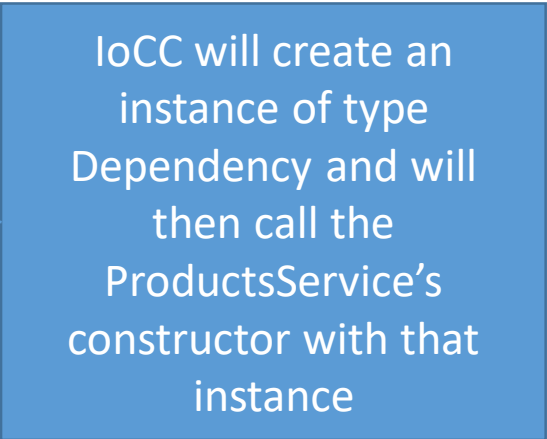
IOCC Analogy – Medical Operation



Applying an IOCC

- There are two ways for the IOCC to determine which dependencies your component needs
- Method A: **Constructor Dependency Injection**
 - Classes that need a dependency can simply state their dependencies in their constructor. The IoCC will examine the constructor and instantiate the dependencies before calling the constructor. And if these dependencies have their own dependencies, then the IoCC will also build them

```
public class ProductService
{
    private Dependency dep;
    0 references | 0 exceptions
    public ProductService(Dependency dep)
    {
        this.dep = dep;
    }
}
```



IoCC will create an instance of type Dependency and will then call the ProductService's constructor with that instance

Applying an IOCC

- Method B: **Property Dependency Injection**

- If the class that the IoCC needs to build has properties that indicate a dependency, then these properties are filled in by the IoCC
- The way a property does this depends on the IoCC (in .NET there are a couple of different IoCC frameworks)
- **In Blazor** you can have the IoCC **inject an instance with the @inject Directive** in your Razor file

```
@page "/productlist"
```

```
@inject IProductsService productService
```

```
<div>  
  @foreach (var product in productService.GetAllProducts())  
  {
```

```
    <div>@product.Name</div>
```

```
    <div>@product.Description</div>
```

```
    <div>@product.UnitPrice</div>
```

```
  }
```

```
</div>
```

```
@code {
```

```
    // TODO: This will throw a NullReferenceException if not instantiated
```

```
    // TODO: This is now required anymore when utilizing the @using directive
```

```
    //IProductsService productService;
```

```
}
```

Injecting a Dependency
with the @inject
Directive

Applying an IOCC

- Method B: **Property Dependency Injection**

- If you're **using code separation**, you can add a property to your class and apply the **[Inject] attribute**

```
public class ProductListBase : ComponentBase
{
    [Inject]
    1 reference | 0 exceptions
    public IProductsService productService { get; set; }
}
```

```
@page "/productlist"
@inherits ProductListBase
```

```
<div>
    @foreach (var product in productService.GetAllProducts())
    {
        <div>@product.Name</div>
        <div>@product.Description</div>
        <div>@product.UnitPrice</div>
    }
</div>
```

ProductList.razor.cs

Add a property to your class and apply the [Inject] attribute

ProductList.razor

Use this property directly in your Razor file without the @inject Directive

Module 6: Services and Dependency Injection

Section 1: Services and Dependency Injection

Lesson: Configuring Dependency Injection

Configuring Dependency Injection

- **Use Case 1:** Dependency is a class
- **Solution:** IoCC can easily know that it needs to create an instance of the class with the class' constructor
- **Use Case 2:** Dependency is an interface, which is generally the case if you are applying the Principle of Dependency Inversion.
 - Which class does it use to create the instance?
- **Solution:** Without additional information it cannot know

Configuring Dependency Injection

- An IoCC has a mapping between interfaces and classes, and it is your job to configure this mapping
- You configure the mapping in your Blazor project's Startup class, just like in ASP.NET Core for Server Side Blazor
- You configure the mapping in your Blazor project's Program class for Blazor WebAssembly

Dependency Scopes

- The extension method selection depends on the lifetime you want to give your dependency

```
public class Startup
```

```
{
```

```
    0 references | 0 exceptions
```

```
    public void ConfigureServices(IServiceCollection services)
```

```
    {
```

```
        services.add
```

```
    }
```

```
    0 references | 0 exce
```

```
    public void C
```

```
    {
```

```
        app.AddCo
```

```
    }
```

```
}
```

```
    AddOptions
```

```
    AddOptions<>
```

```
    AddScoped
```

```
    AddScoped<>
```

```
    AddSingleton
```

```
    AddSingleton<>
```

```
    AddTransient
```

```
    AddTransient<>
```

```
    [Icons]
```

```
void System.Collections.Generic.ICollection<ServiceDescriptor>.Add(ServiceDescriptor item)  
Adds an item to the System.Collections.Generic.ICollection<T>.
```

```
    app.Services.AddSingleton<IConfiguration>(app.Configuration);
```

```
    app.Services.AddTransient<IConfiguration>();
```

Singleton Dependencies

- Singleton classes are classes that **only have one instance**
- They are **typically used to manage some global state**. For example, you could have a class that keeps track of how many times people have clicked a certain product
- Singleton classes **can also be classes that don't have any state, that only have behavior** (utility classes such as one that does conversions between imperial and metric units)
- You configure dependency injection to reuse the same instance all the time with the **AddSingleton extension method**

Scoped Dependencies

- Blazor WebAssembly apps don't currently have a concept of DI scopes. Scoped-registered services behave like Singleton services
- However, the Blazor Server hosting model supports the Scoped lifetime. In Blazor Server apps, a scoped service registration is scoped to the connection. For this reason, using scoped services is preferred for services that should be scoped to the current user, even if the current intent is to run client-side in the browser

Transient Dependencies

- **Each time** an instance needs to be created by the IoCC it will create a **fresh instance**
- The IoCC will also **Dispose of the instance** (when your class implements the **IDisposable interface**) **when it is no longer needed**
- You configure dependency injection to use transient instances with the **AddTransient extension method**

Disposing Dependencies

- One of the advantages of dependency injection is that it takes care of calling the Dispose method of instances that implement IDisposable
- For **singleton** instances, **cleanup** happens **at the end of the program**
- For scoped instances, **cleanup** happens **at the end of the request**
- For transient instances, **cleanup** happens when the component is removed from the UI
- In general, if your classes implement IDisposable correctly, you don't have to take care of anything else

Default Services

- Default services are automatically added to the app's service collection

Service	Lifetime	Description
HttpClient	Singleton	<p>Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.</p> <p>The instance of <code>HttpClient</code> in a Blazor WebAssembly app uses the browser for handling the HTTP traffic in the background.</p> <p>Blazor Server apps don't include an <code>HttpClient</code> configured as a service by default. Provide an <code>HttpClient</code> to a Blazor Server app.</p> <p>For more information, see Call a web API from ASP.NET Core Blazor.</p>
<code>IJSRuntime</code>	Singleton	<p>Represents an instance of a JavaScript runtime where JavaScript calls are dispatched. For more information, see ASP.NET Core Blazor JavaScript interop.</p>
<code>NavigationManager</code>	Singleton	<p>Contains helpers for working with URIs and navigation state. For more information, see URI and navigation state helpers.</p>

Demo: Dependency Injection

Module Summary

- In this module, you learned about:
 - Dependency Injection Basics
 - Using Dependency Inversion Principle
 - Adding Dependency Injection
 - Configuring Dependency Injection
 - Dependency Scopes



Lab 6: Services and Dependency Injection



References

- [Microsoft Docs](#)
- [Blazor University](#)

