Blazor

Module 6: Dependency Injection

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

**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.

© 2020 Microsoft Corporation. All rights reserved.

**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, SQL Server, 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.

Contents

[Lab 6: Dependency Injection 1](#_Toc38805889)

[Exercise 1: Dependency Injection 3](#_Toc38805890)

# Lab 6: Dependency Injection

#### Introduction

The aim of this exercise is to explore implementing an application using the new application model introduced under ASP.Net Core 3 called Blazor.

Important

**Blazor WebAssembly in preview**

Blazor Server is supported in ASP.NET Core 3.0. Blazor WebAssembly is in preview for ASP.NET Core 3.1.

#### Objectives

After completing this lab, you will be able to:

* Learn how to manage state across components

#### Prerequisites

None

#### Scenario

In this scenario, we will explore implementing an online Pizza Delivery application.

#### System Requirements

To complete this lab, you need:

* Follow the instructions found [on this page](https://docs.microsoft.com/en-us/aspnet/core/blazor/get-started?view=aspnetcore-3.1&tabs=visual-studio) to get started

#### Estimated Time to Complete This Lab

20 minutes

Exercise 1: Dependency Injection

#### Objectives

In this exercise, you will:

* Learn how to manage state across components

#### Scenario

In this exercise we will revisit some of the code we have already written and try to make it nicer. We will also talk more about how events cause the UI to update.

Task 1: Adding A Navigation Link

1. Start by opening the solution file **BlazingPizza.sln** located under **\Labs\Module 06 - Dependency Injection\Begin**.
2. You might have noticed this already, but our application has a bug! Since we are storing the list of pizzas in the current order on the Index component, the user's state can be lost if the user leaves the index page. To see this in action, add a pizza to the current order (don't place the order yet) - then navigate to the MyOrders page and back to Index. When you get back, you will notice the order is empty!

We are going to fix this bug by introducing something we've dubbed the AppState pattern. The basics are that you want to add an object to the DI container that you will use to coordinate state between related components. Because the AppState object is managed by the DI container, it can outlive the components and hold on to state even when the UI is changing a lot. Another benefit of the AppState pattern is that it leads to greater separation between presentation (components) and business logic.

1. Create a new class called OrderState in the *BlazingPizza.Client* root directory - and register it as a scoped service in the DI container. In Blazor WebAssembly applications, services are registered in the *Program.cs* class via the Main method. Add the service just before the call to await builder.Build().RunAsync();.

public static async Task Main(string[] args)

{

…

builder.Services.AddScoped<OrderState>();

…

}

**Note:** The reason why we chose scoped over singleton is for symmetry with a server-side-components application. Singleton usually means for all users, whereas scoped means for the current unit-of-work.

1. Now that this type is registered in DI, we can @inject it into the Index page.

@page "/"

@inject HttpClient HttpClient

@inject OrderState OrderState

@inject NavigationManager NavigationManager

Recall that @inject is a convenient shorthand to both retrieve something from DI by type, and to define a property of that type. You can test this now by running the app again. If you try to inject something that isn't found in the DI container, then it will throw an exception and the Index will fail to come up.

1. Now, let's add properties and methods to this class that will represent and manipulate the state of an Order and a Pizza.

Move the configuringPizza, showingConfigureDialog and order to be properties on the OrderState class. I like to make them private set so they can only be manipulated via methods on OrderState.

public class OrderState

{

public bool ShowingConfigureDialog { get; private set; }

public Pizza ConfiguringPizza { get; private set; }

public Order Order { get; private set; } = new Order();

}

1. Now let's move some of the methods from the Index to OrderState. We won't move PlaceOrder into OrderState because that triggers a navigation, so instead we'll just add a ResetOrder method.

public void ShowConfigurePizzaDialog(PizzaSpecial special)

{

ConfiguringPizza = new Pizza()

{

Special = special,

SpecialId = special.Id,

Size = Pizza.DefaultSize,

Toppings = new List<PizzaTopping>(),

};

showingConfigureDialog = true;

}

public void CancelConfigurePizzaDialog()

{

ConfiguringPizza = null;

ShowingConfigureDialog = false;

}

public void ConfirmConfigurePizzaDialog()

{

Order.Pizzas.Add(ConfiguringPizza);

ConfiguringPizza = null;

ShowingConfigureDialog = false;

}

public void ResetOrder()

{

Order = new Order();

}

public void RemoveConfiguredPizza(Pizza pizza)

{

Order.Pizzas.Remove(pizza);

}

Remember to remove the corresponding methods from Index.razor. You must also remember to remove the order, configuringPizza, and showingConfigureDialog fields entirely from Index.razor, since you'll be getting the state data from the injected OrderState.

1. At this point it should be possible to get the Index component compiling again by updating references to refer to various bits attached to OrderState. For example, the remaining PlaceOrder method in Index.razor may look something like this:

async Task PlaceOrder()

{

var response = await HttpClient.PostAsJsonAsync<Order>("orders",   
 Order);

var newOrderId = await   
 response.Content.ReadFromJsonAsync<int>();

OrderState.ResetOrder();

NavigationManager.NavigateTo($"myorders/{newOrderId}");

}

Feel free to create convenience properties for things like OrderState.Order or OrderState.Order.Pizzas if it feels better to you that way. Try this out and verify that everything still works. Verify that you've fixed the original bug: you can now add some pizzas, navigate to "My orders", navigate back, and your order has now been saved.

This is a good opportunity to explore how state changes and rendering work in Blazor, and how EventCallback solves some common problems. The detail of what is happening now became more complicated with OrderState involved. EventCallback tells Blazor to dispatch the event notification (and rendering) to the component that defined the event handler. If the event handler is not defined by a component (OrderState) then it will substitute the component that *hooked up* the event handler (Index).