Blazor

Module 4: Events and Data Binding

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 4: Events and Data Binding 1](#_Toc29923018)

[Exercise 1: Event Handling and Data Binding 3](#_Toc29923019)

# Lab 4: Events and Data Binding

#### 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.

#### Objectives

After completing this lab, you will be able to:

* Learn how to handle DOM UI Events
* Learn how to pass Component Parameters
* Learn how to implement Data Binding
* Learn how to implement Component Events

#### Prerequisites

None

#### Scenario

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

#### System Requirements

* 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

60 minutes

Exercise 1: Event Handling and Data Binding

#### Objectives

In this exercise, you will:

* Learn how to handle DOM UI Events
* Learn how to pass Component Parameters
* Learn how to implement Data Binding
* Learn how to implement Component Events

#### Scenario

We will update the pizza store app to enable users to customize their pizzas and add them to their order.

Task 1: Handle DOM UI Events

1. When the user clicks a pizza special, a pizza customization dialog should pop up to allow the user to customize their pizza and add it to their order. To handle DOM UI events in a Blazor app, you specify which event you want to handle using the corresponding HTML attribute and then specify the C# delegate you want called. The delegate may optionally take an event specific argument, but it is not required.

In BlazingPizza.Client/Pages/Index.razor add the following @onclick handler to the list item for each pizza special:

@foreach (var special in specials)

{

<li @onclick="() => Console.WriteLine(special.Name)" style="background-image: url('@special.ImageUrl')">

<div class="pizza-info">

<span class="title">@special.Name</span>

@special.Description

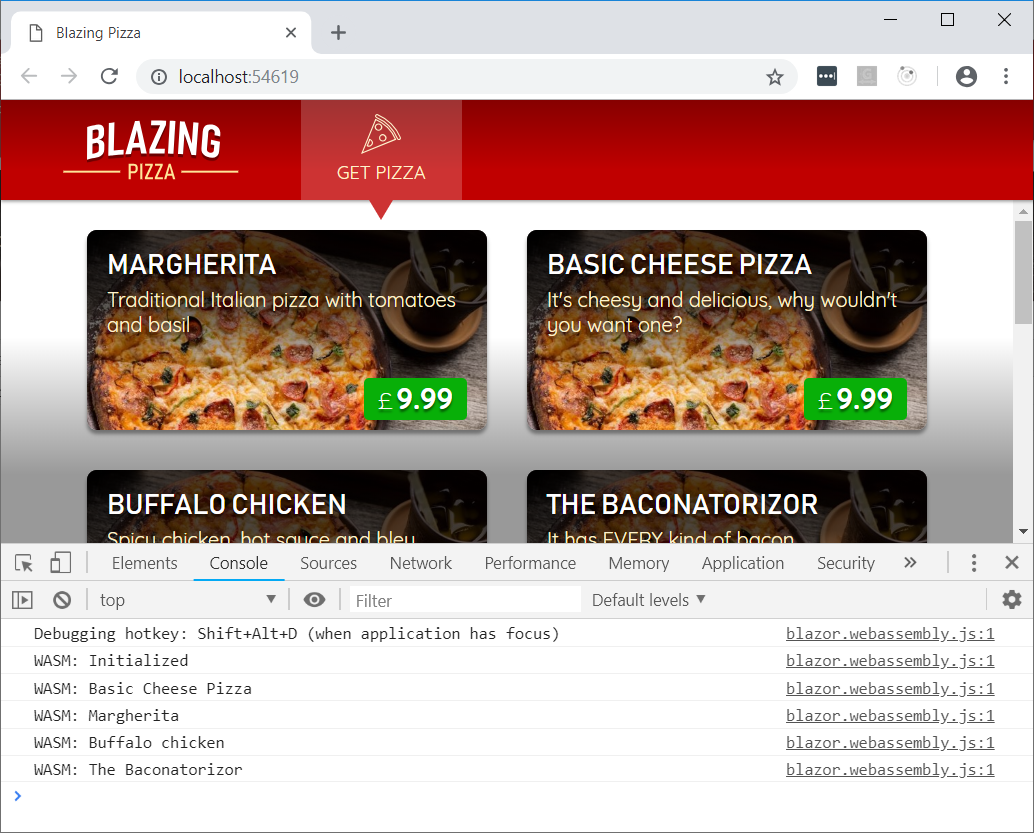
<span class="price">@special.GetFormattedBasePrice()</span>

</div>

</li>

}

1. Run the app and check that the pizza name is written to the browser console whenever a pizza is clicked.

[](https://user-images.githubusercontent.com/1874516/51804286-ce965000-2256-11e9-87fc-a8770ccc70d8.png)

1. The @ symbol is used in Razor files to indicate the start of C# code. Surround the C# code with parentheses if needed to clarify where the C# code begins and ends.
2. Update the @code block in Index.razor to add some additional fields for tracking the pizza being customized and whether the pizza customization dialog is visible.

List<PizzaSpecial> specials;

Pizza configuringPizza;

bool showingConfigureDialog;

1. Add a ShowConfigurePizzaDialog method to the @code block for handling when a pizza special is clicked.

void ShowConfigurePizzaDialog(PizzaSpecial special)

{

configuringPizza = new Pizza()

{

Special = special,

SpecialId = special.Id,

Size = Pizza.DefaultSize,

Toppings = new List<PizzaTopping>(),

};

showingConfigureDialog = true;

}

1. Update the @onclick handler to call the ShowConfigurePizzaDialog method instead of Console.WriteLine.

<li @onclick="() => ShowConfigurePizzaDialog(special)" style="background-image: url('@special.ImageUrl')">

Task 2: Passing Component Parameters

1. Now we need to implement the pizza customization dialog so we can display it when the user selects a pizza. The pizza customization dialog will be a new component that lets you specify the size of your pizza and what toppings you want, shows the price, and lets you add the pizza to your order. Add a **ConfigurePizzaDialog.razor** file under the **Shared** directory. Since this component is not a separate page, it does not need the @page directive.

Important

**In Visual Studio, you can right-click the Shared directory in Solution Explorer, then choose Add -> New Item, then use the Razor Component item template.**

1. The ConfigurePizzaDialog should have a Pizza parameter that specifies the pizza being configured. Component parameters are defined by adding a writable property to the component decorated with the [Parameter] attribute. Add a @code block to the ConfigurePizzaDialog with the following Pizza parameter:

@code {

[Parameter] public Pizza Pizza { get; set; }

}

Important

**Component parameter values need to have a setter and be declared public because they get set by the framework. However, they should only be set by the framework as part of the rendering process. Do not write code that overwrites these parameter values from outside the component, because then your component's state will be out of sync with its render output.**

1. Add the following basic markup for the ConfigurePizzaDialog:

<div class="dialog-container">

<div class="dialog">

<div class="dialog-title">

<h2>@Pizza.Special.Name</h2>

@Pizza.Special.Description

</div>

<form class="dialog-body"></form>

<div class="dialog-buttons">

<button class="btn btn-secondary mr-auto">Cancel</button>

<span class="mr-center">

Price: <span class="price">@(Pizza.GetFormattedTotalPrice())</span>

</span>

<button class="btn btn-success ml-auto">Order ></button>

</div>

</div>

</div>

1. Update Pages/Index.razor to show the ConfigurePizzaDialog when a pizza special has been selected. The ConfigurePizzaDialog is styled to overlay the current page, so it doesn't really matter where you put this code block.

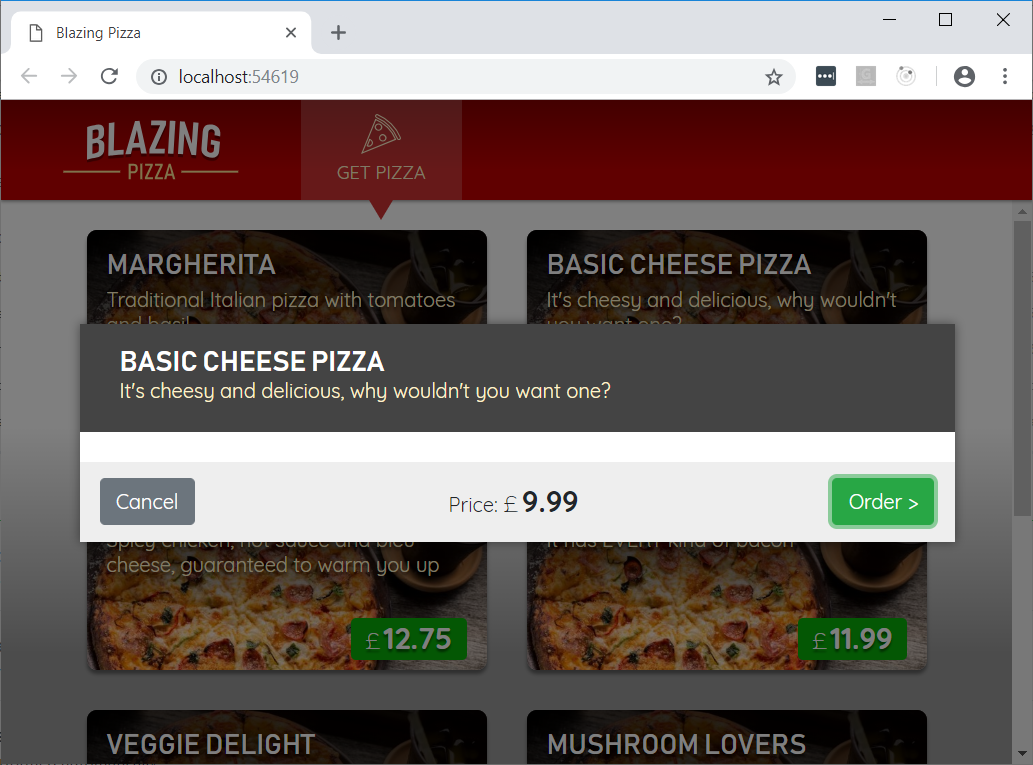
@if (showingConfigureDialog)

{

<ConfigurePizzaDialog Pizza="configuringPizza" />

}

1. Run the app and select a pizza special to see the skeleton of the ConfigurePizzaDialog.

[](https://user-images.githubusercontent.com/1874516/51804297-e8d02e00-2256-11e9-85a6-da0becf7130d.png)

Unfortunately, at this point there is no functionality in place to close the dialog. We will add that shortly. Let us get to work on the dialog itself.

Task 3: Data Binding

1. Next you will allow the user to specify the size of their pizza. Add markup to the body of ConfigurePizzaDialog for a slider that lets the user specify the pizza size. This should replace the existing <form class="dialog-body"></form> element:

<form class="dialog-body">

<div>

<label>Size:</label>

<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" />

<span class="size-label">

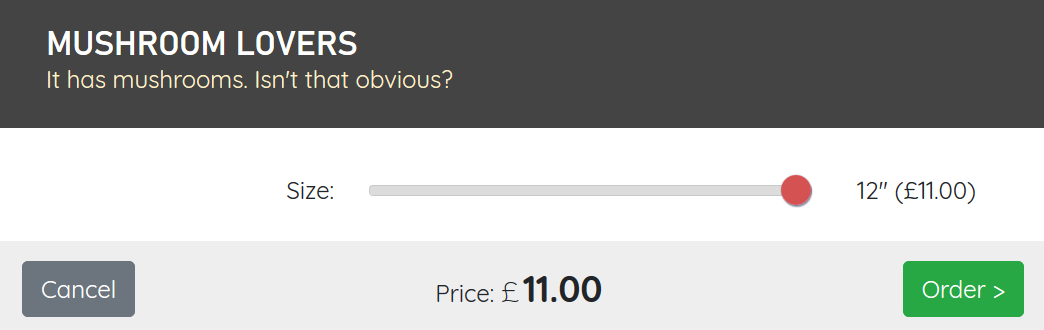
@(Pizza.Size)" (£@(Pizza.GetFormattedTotalPrice()))

</span>

</div>

</form>

1. Run the app and select a pizza special. Now the dialog shows a slider that can be used to change the pizza size. However, it doesn't do anything right now.

[](https://user-images.githubusercontent.com/1430011/57576985-eff40400-7421-11e9-9a1b-b22d96c06bcb.png)

1. We want to make it so the value of the Pizza.Size will reflect the value of the slider. When the dialog opens, the slider gets its value from Pizza.Size. Moving the slider should update the pizza size stored in Pizza.Size accordingly. This concept is called two-way binding.

If you wanted to implement two-way binding manually, you could do so by combining value and @onchange, as in the following code (which you don't actually need to put in your application, because there's an easier solution):

<input

type="range"

min="@Pizza.MinimumSize"

max="@Pizza.MaximumSize"

step="1"

value="@Pizza.Size"

@onchange="(ChangeEventArgs e) => Pizza.Size = int.Parse((string) e.Value)" />

In Blazor you can use the @bind directive attribute to specify a two-way binding with this behavior. The equivalent markup using @bind looks like this:

<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" />

But if we use @bind with no further changes, the behavior isn't exactly what we want. Give it a try and see how it behaves. The update event only fires after the slider is released.

We would prefer to see updates as the slider is moved. Data binding in Blazor allows for this by letting you specify what event triggers a change using the syntax @bind:<eventname>. So, to bind using the oninput event instead do this:

<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" @bind:event="oninput" />

The pizza size should now update as you move the slider.

Task 4: Adding Additional Event Handlers

1. The user should also be able to select additional toppings on ConfigurePizzaDialog. Add a list property for storing the available toppings. Initialize the list of available toppings by making an HTTP GET request to the /toppings API.

@inject HttpClient HttpClient

<div class="dialog-container">

...

</div>

@code {

List<Topping> toppings;

[Parameter] public Pizza Pizza { get; set; }

protected override async Task OnInitializedAsync()

{

toppings = await HttpClient.GetFromJsonAsync<List<Topping>>("toppings");

}

}

1. Add the following markup in the dialog body for displaying a drop down list with available toppings, followed by the set of selected toppings. Put this inside the <form class="dialog-body">, below the existing <div>.

<div>

<label>Extra Toppings:</label>

@if (toppings == null)

{

<select class="custom-select" disabled>

<option>(loading...)</option>

</select>

}

else if (Pizza.Toppings.Count >= 6)

{

<div>(maximum reached)</div>

}

else

{

<select class="custom-select" @onchange="ToppingSelected">

<option value="-1" disabled selected>(select)</option>

@for (var i = 0; i < toppings.Count; i++)

{

<option value="@i">@toppings[i].Name - (£@(toppings[i].GetFormattedPrice()))</option>

}

</select>

}

</div>

<div class="toppings">

@foreach (var topping in Pizza.Toppings)

{

<div class="topping">

@topping.Topping.Name

<span class="topping-price">@topping.Topping.GetFormattedPrice()</span>

<button type="button" class="delete-topping" @onclick="() => RemoveTopping(topping.Topping)">x</button>

</div>

}

</div>

1. Also add the following event handlers for topping selection and removal:

void ToppingSelected(ChangeEventArgs e)

{

if (int.TryParse((string)e.Value, out var index) && index >= 0)

{

AddTopping(toppings[index]);

}

}

void AddTopping(Topping topping)

{

if (Pizza.Toppings.Find(pt => pt.Topping == topping) == null)

{

Pizza.Toppings.Add(new PizzaTopping() { Topping = topping });

}

}

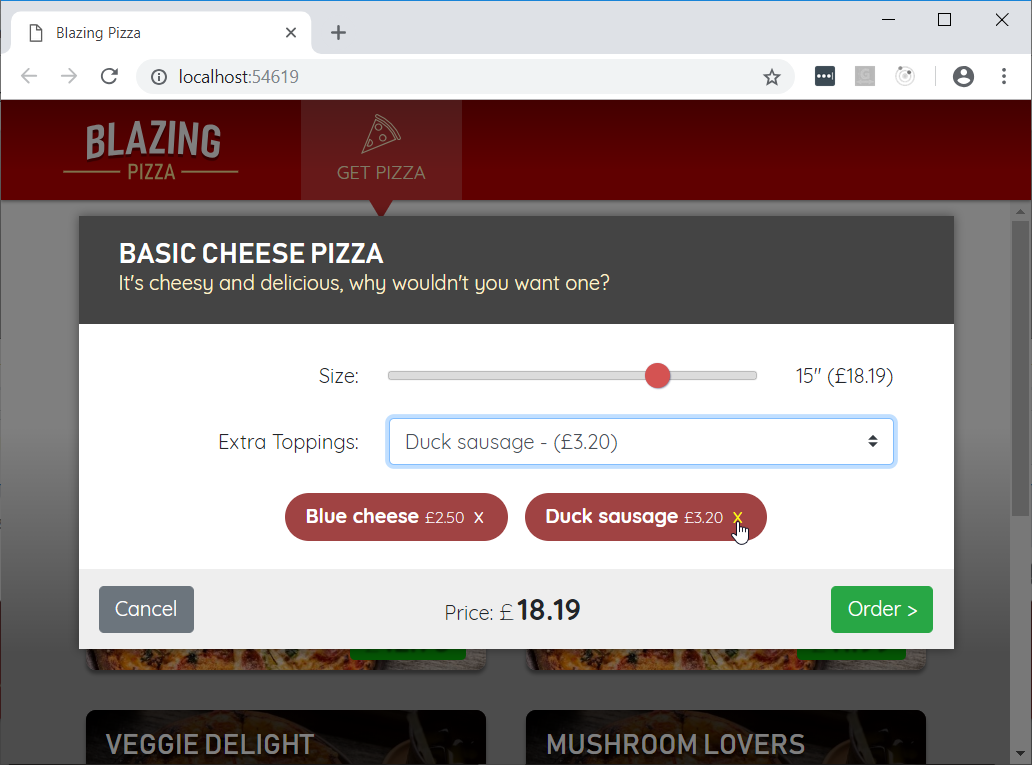
void RemoveTopping(Topping topping)

{

Pizza.Toppings.RemoveAll(pt => pt.Topping == topping);

}

You should now be able to add and remove toppings.

[](https://user-images.githubusercontent.com/1874516/51805012-f50cb900-225f-11e9-8642-4e6d34a48c3f.png)

Task 5: Component Events

1. The Cancel and Order buttons don't do anything yet. We need some way to communicate to the Index component when the user adds the pizza to their order or cancels. We can do that by defining component events. Component events are callback parameters that parent components can subscribe to. Add two parameters to the ConfigurePizzaDialog component: OnCancel and OnConfirm. Both parameters should be of type EventCallback.

[Parameter] public EventCallback OnCancel { get; set; }

[Parameter] public EventCallback OnConfirm { get; set; }

1. Add @onclick event handlers to the ConfigurePizzaDialog that trigger the OnCancel and OnConfirm events.

<div class="dialog-buttons">

<button class="btn btn-secondary mr-auto" @onclick="OnCancel">Cancel</button>

<span class="mr-center">

Price: <span class="price">@(Pizza.GetFormattedTotalPrice())</span>

</span>

<button class="btn btn-success ml-auto" @onclick="OnConfirm">Order ></button>

</div>

1. In the Index component add an event handler for the OnCancel event that hides the dialog and wires it up to the ConfigurePizzaDialog.

<ConfigurePizzaDialog Pizza="configuringPizza" OnCancel="CancelConfigurePizzaDialog" />

void CancelConfigurePizzaDialog()

{

configuringPizza = null;

showingConfigureDialog = false;

}

Now, what happens when you click the dialog cancel button is that Index.CancelConfigurePizzaDialog will execute, and then the Index component will render itself. Since showingConfigureDialog is now false the dialog will not be displayed.

Normally what happens when you trigger an event (like clicking the cancel button) is that the component that defined the event handler delegate will rerender. You could define events using any delegate type like Action or Func<string, Task>. Sometimes you want to use an event handler delegate that doesn't belong to a component - if you used a normal delegate type to define the event then nothing will be rendered or updated.

EventCallback is a special type that is known to the compiler that resolves some of these issues. It tells the compiler to dispatch the event to the component that contains the event handler logic. EventCallback has a few more tricks up its sleeve, but for now just remember that using EventCallback makes your component smart about dispatching events to the right place.

1. Run the app and verify that the dialog now disappears when the Cancel button is clicked.
2. When the OnConfirm event is fired, the customized pizza should be added to the user's order. Add an Order field to the Index component to track the user's order.

List<PizzaSpecial> specials;

Pizza configuringPizza;

bool showingConfigureDialog;

Order order = new Order();

In the Index component add an event handler for the OnConfirm event that adds the configured pizza to the order and wire it up to the ConfigurePizzaDialog.

<ConfigurePizzaDialog

Pizza="configuringPizza"

OnCancel="CancelConfigurePizzaDialog"

OnConfirm="ConfirmConfigurePizzaDialog" />

void ConfirmConfigurePizzaDialog()

{

order.Pizzas.Add(configuringPizza);

configuringPizza = null;

showingConfigureDialog = false;

}

1. Run the app and verify the dialog now disappears when the Order button is clicked. We can't see yet that a pizza was added to the order because there's no UI that shows this information. We will address that next.
2. Next, we need to display the configured pizzas in the current order, calculate the total price, and provide a way to place the order.

Create a new ConfiguredPizzaItem component for displaying a configured pizza. It takes two parameters: the configured pizza, and an event for when the pizza was removed.

<div class="cart-item">

<a @onclick="OnRemoved" class="delete-item">x</a>

<div class="title">@(Pizza.Size)" @Pizza.Special.Name</div>

<ul>

@foreach (var topping in Pizza.Toppings)

{

<li>+ @topping.Topping.Name</li>

}

</ul>

<div class="item-price">

@Pizza.GetFormattedTotalPrice()

</div>

</div>

@code {

[Parameter] public Pizza Pizza { get; set; }

[Parameter] public EventCallback OnRemoved { get; set; }

}

1. Add the following markup to the Index component just below the main <div> to add a right-side pane for displaying the configured pizzas in the current order.

<div class="sidebar">

@if (order.Pizzas.Any())

{

<div class="order-contents">

<h2>Your order</h2>

@foreach (var configuredPizza in order.Pizzas)

{

<ConfiguredPizzaItem Pizza="configuredPizza" OnRemoved="() => RemoveConfiguredPizza(configuredPizza)" />

}

</div>

}

else

{

<div class="empty-cart">Choose a pizza<br>to get started</div>

}

<div class="order-total @(order.Pizzas.Any() ? "" : "hidden")">

Total:

<span class="total-price">@order.GetFormattedTotalPrice()</span>

<button class="btn btn-warning" disabled="@(order.Pizzas.Count == 0)" @onclick="PlaceOrder">

Order >

</button>

</div>

</div>

1. Also add the following event handlers to the Index component for removing a configured pizza from the order and submitting the order.

void RemoveConfiguredPizza(Pizza pizza)

{

order.Pizzas.Remove(pizza);

}

async Task PlaceOrder()

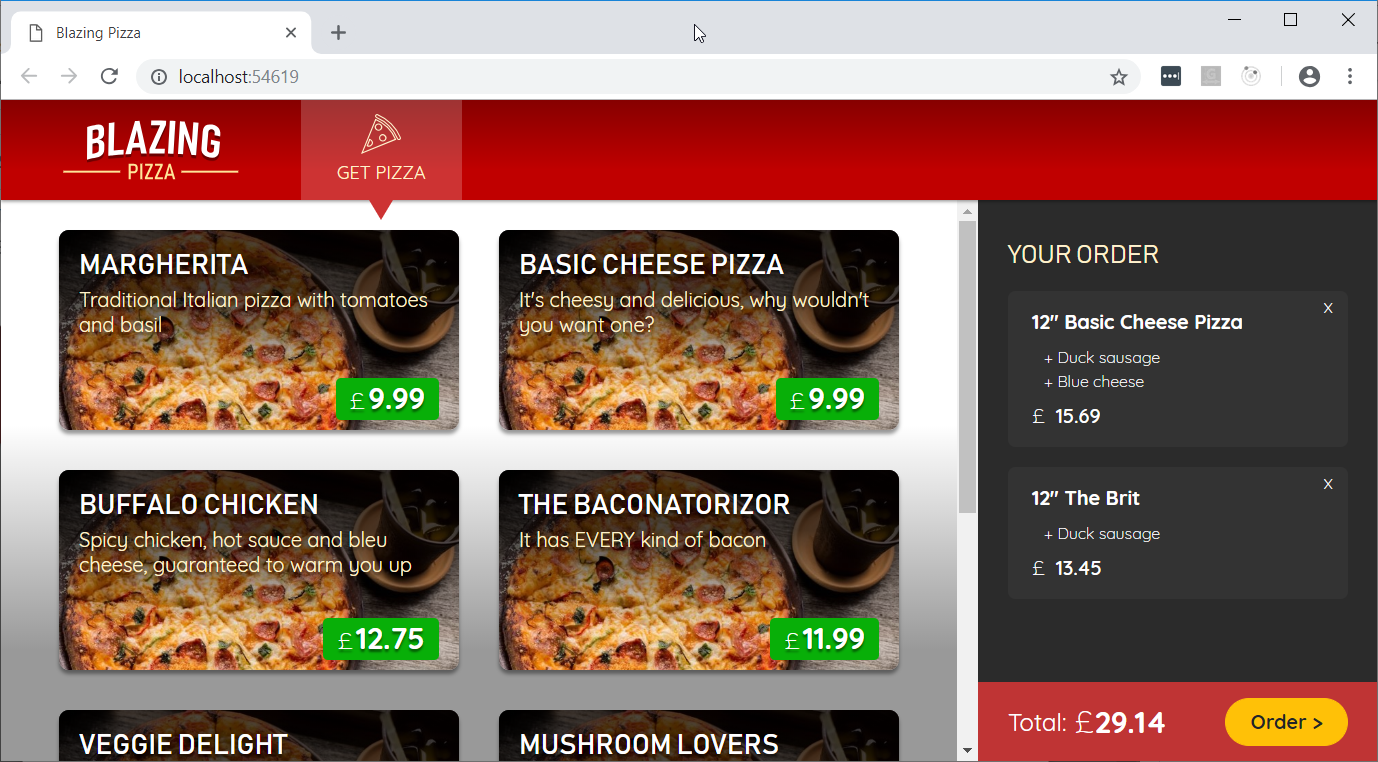
{

await HttpClient.PostAsJsonAsync("orders", order);

order = new Order();

}

You should now be able to add and remove configured pizzas from the order and submit the order.

[](https://user-images.githubusercontent.com/1874516/51805192-59c91300-2262-11e9-9b6f-d8f2d606feda.png)

Even though the order was successfully added to the database, there is nothing in the UI yet that indicates this happened. That is what we'll address in the next session.