Blazor

Module 5: Routing

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 5: Routing 1](#_Toc29922912)

[Exercise 1: Routing 3](#_Toc29922913)

# Lab 5: Routing

#### 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:

* Introduce routing to the application

#### 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

60 minutes

Exercise 1: Routing

#### Objectives

In this exercise, you will:

* Introduce routing to the application

#### Scenario

Your customers can order pizzas, but so far have no way to see the status of their orders. In this Exercise you'll implement a "My orders" page that lists multiple orders, plus an "Order details" view showing the contents and status of an individual order.

Task 1: Adding A Navigation Link

1. Start by opening the solution file **BlazingPizza.sln** located under **\Labs\Module 05 - Routing\Begin**.
2. Open **Shared/MainLayout.razor**. As an experiment, let's try adding a new link element without using NavLink. Add a plain HTML <a> tag pointing to myorders:

<div class="top-bar">

(leave existing content in place)

<a href="myorders" class="nav-tab">

<img src="img/bike.svg" />

<div>My Orders</div>

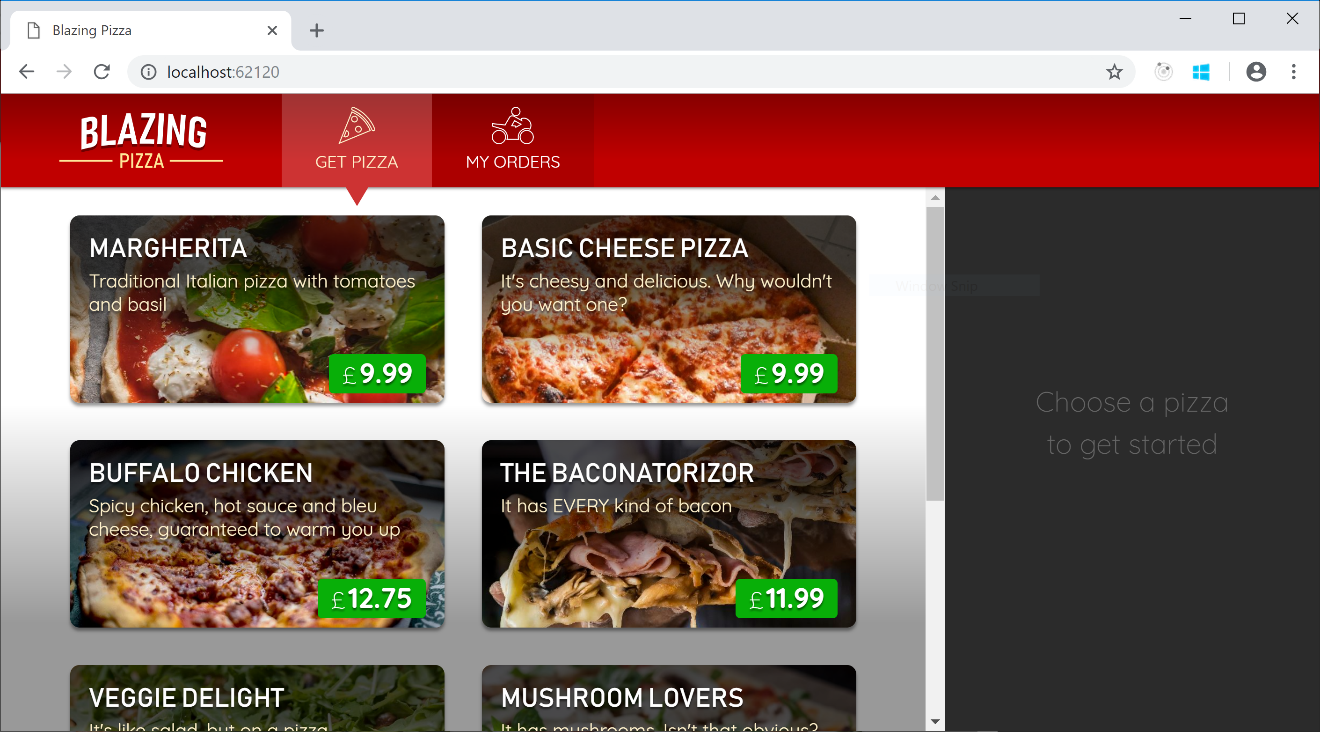
</a>

</div>

Important

**Notice how the URL we are linking to does not start with a /. If you linked to /myorders, it would appear to work the same, but if you ever wanted to deploy the app to a non-root URL the link would break. The <base href="/"> tag in index.html specifies the prefix for all non-slash-prefixed URLs in the app, regardless of which component renders them.**

1. If you run the app now, you'll see the link (MY ORDERS), styled as expected:

[](https://user-images.githubusercontent.com/1101362/51804403-60528d00-2258-11e9-8d2b-ab00d33c74cb.png)

This shows it's not strictly necessary to use <NavLink>. We'll see the reason to use it momentarily.

1. If you click "My Orders", you'll end up on a page that says "Page not found". Obviously this is because you haven't yet added anything that matches the URL myorders. But if you're watching really carefully, you might notice that on this occasion it's not just doing client-side (SPA-style) navigation, but instead is doing a full-page reload.   
   What's really happening is this:
   1. You click on the link to myorders
   2. Blazor, running on the client, tries to match this to a client-side component based on @page directive attributes.
   3. However, no match is found, so Blazor falls back on a full-page load navigation in case the URL is meant to be handled by server-side code.
   4. However, the server doesn't have anything that matches this either, so it falls back on rendering the client-side Blazor application.
   5. This time, Blazor sees that nothing matches on either client or server, so it falls back on rendering the NotFound block from your App.razor component.

If you want to, try changing the content in the NotFound block in App.razor to see how you can customize this message.   
As you can guess, we will make the link actually work by adding a component to match this route. Create a file in the Pages folder called MyOrders.razor, with the following content:

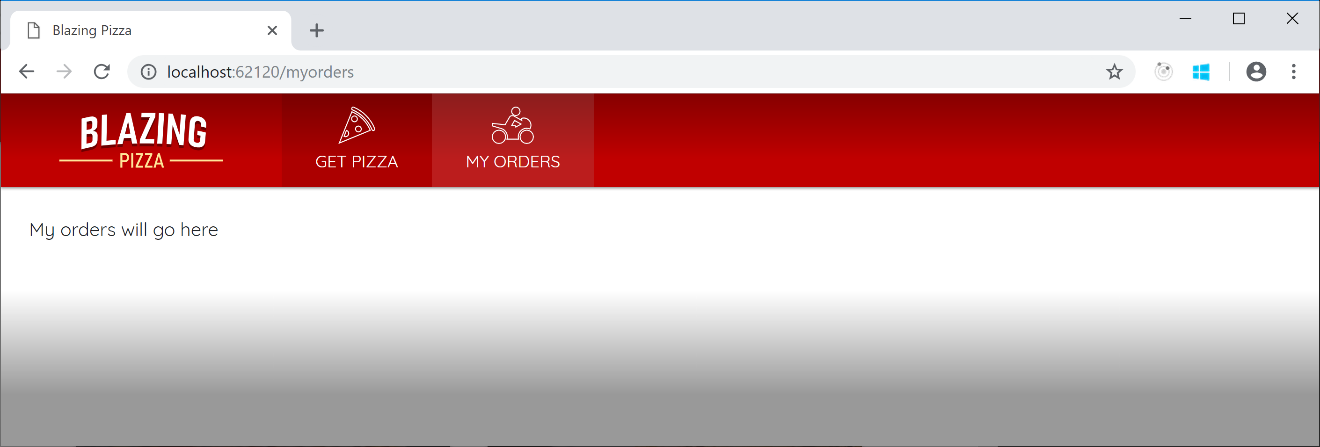
@page "/myorders"

<div class="main">

My orders will go here

</div>

1. Now when you run the app, you'll be able to visit this page. Also notice that this time, no full-page load occurs when you navigate, because the URL is matched entirely within the client-side SPA. As such, navigation is instantaneous.

[](https://user-images.githubusercontent.com/1101362/51804512-c855a300-2259-11e9-8770-b4b8c318ba9d.png)

1. Look closely at the top bar. Notice that when you're on "My orders", the link isn't highlighted in yellow. How can we highlight links when the user is on them? By using a <NavLink> instead of a plain <a> tag. The only thing a NavLink does is toggle its own active CSS class depending on whether it matches the current navigation state. Replace the <a> tag you just added in MainLayout with the following (which is identical apart from the tag name):

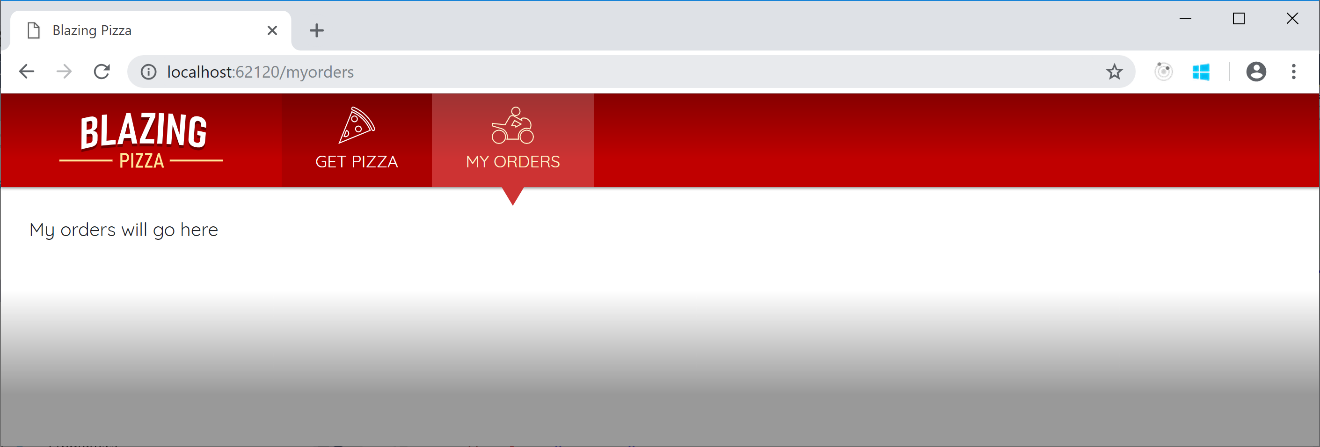
<NavLink href="myorders" class="nav-tab">

<img src="img/bike.svg" />

<div>My Orders</div>

</NavLink>

1. Now you'll see the links are correctly highlighted according to navigation state:

[](https://user-images.githubusercontent.com/1101362/51804583-ca6c3180-225a-11e9-86cb-58a5a469e3f7.png)

1. Switch back to the MyOrders component code. Once again we're going to inject an HttpClient so that we can query the backend for data. Add the following under the @page directive line:

@inject HttpClient HttpClient

1. Then add a @code block that makes an asynchronous request for the data we need:

@code {

List<OrderWithStatus> ordersWithStatus;

protected override async Task OnParametersSetAsync()

{

ordersWithStatus = await HttpClient.GetFromJsonAsync<List<OrderWithStatus>>("orders");

}

}

1. Let's make the UI display different output in three different cases:

* While we're waiting for the data to load
* If it turns out that the user has never placed any orders
* If the user has placed one or more orders

It's simple to express this using @if/else blocks in Razor code. Update the markup inside your component as follows:

<div class="main">

@if (ordersWithStatus == null)

{

<text>Loading...</text>

}

else if (ordersWithStatus.Count == 0)

{

<h2>No orders placed</h2>

<a class="btn btn-success" href="">Order some pizza</a>

}

else

{

<text>TODO: show orders</text>

}

</div>

Perhaps some parts of this code aren't obvious, so let's point out a few things:

* <text> is not an HTML element at all. Nor is it a component. Once the MyOrderscomponent is compiled, the <text> tag won't exist in the result at all.
* If <a href=""> (with an empty string for href) surprises you, remember that the browser will prefix the <base href="/"> value to all non-slash-prefixed URLs. So, an empty string is the correct way to link to the client app's root URL.
* The asynchronous flow we've implemented above means the component will render twice: once before the data has loaded (displaying "Loading.."), and then once afterwards (displaying one of the other two outputs).
* If you want to reset your database to see the "no orders" case, simply delete pizza.db from the Server project and reload the page in your browser.

1. Now that we have all the data we need; we can use Razor syntax to render an HTML grid. Replace the <text>TODO: show orders</text> code with the following. It looks like a lot of code, but there's nothing special here. It simply uses a @foreach to iterate over the ordersWithStatus and outputs a <div> for each one. The net result is as follows:

<div class="list-group orders-list">

@foreach (var item in ordersWithStatus)

{

<div class="list-group-item">

<div class="col">

<h5>@item.Order.CreatedTime.ToLongDateString()</h5>

Items:

<strong>@item.Order.Pizzas.Count()</strong>;

Total price:

<strong>£@item.Order.GetFormattedTotalPrice()</strong>

</div>

<div class="col">

Status: <strong>@item.StatusText</strong>

</div>

<div class="col flex-grow-0">

<a href="myorders/@item.Order.OrderId" class="btn btn-success">

Track &gt;

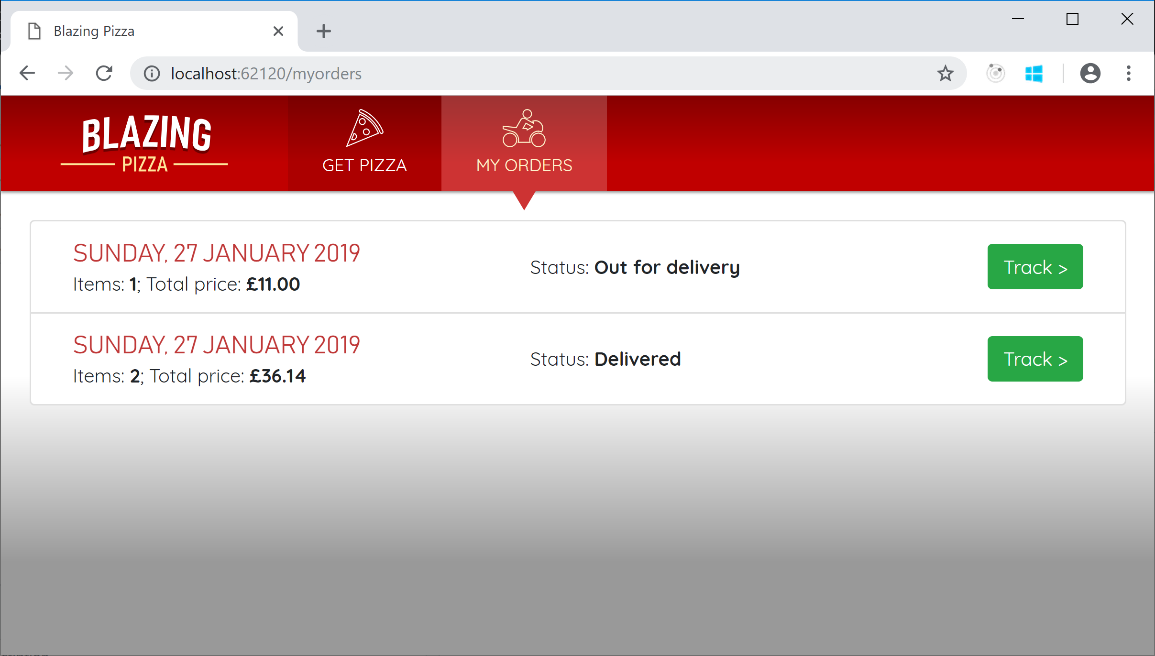
</a>

</div>

</div>

}

</div>

[](https://user-images.githubusercontent.com/1101362/51804902-300ded00-225e-11e9-85b7-6aa2ac764123.png)

1. If you click on the "Track" link buttons next to an order, the browser will attempt to navigation to myorders/<id> (e.g., http://example.com/myorders/37). Currently this will result in a "Page not found" message because no component matches this route.

Once again we'll add a component to handle this. In the Pages directory, create a file called OrderDetails.razor, containing:

@page "/myorders/{orderId:int}"

<div class="main">

TODO: Show details for order @OrderId

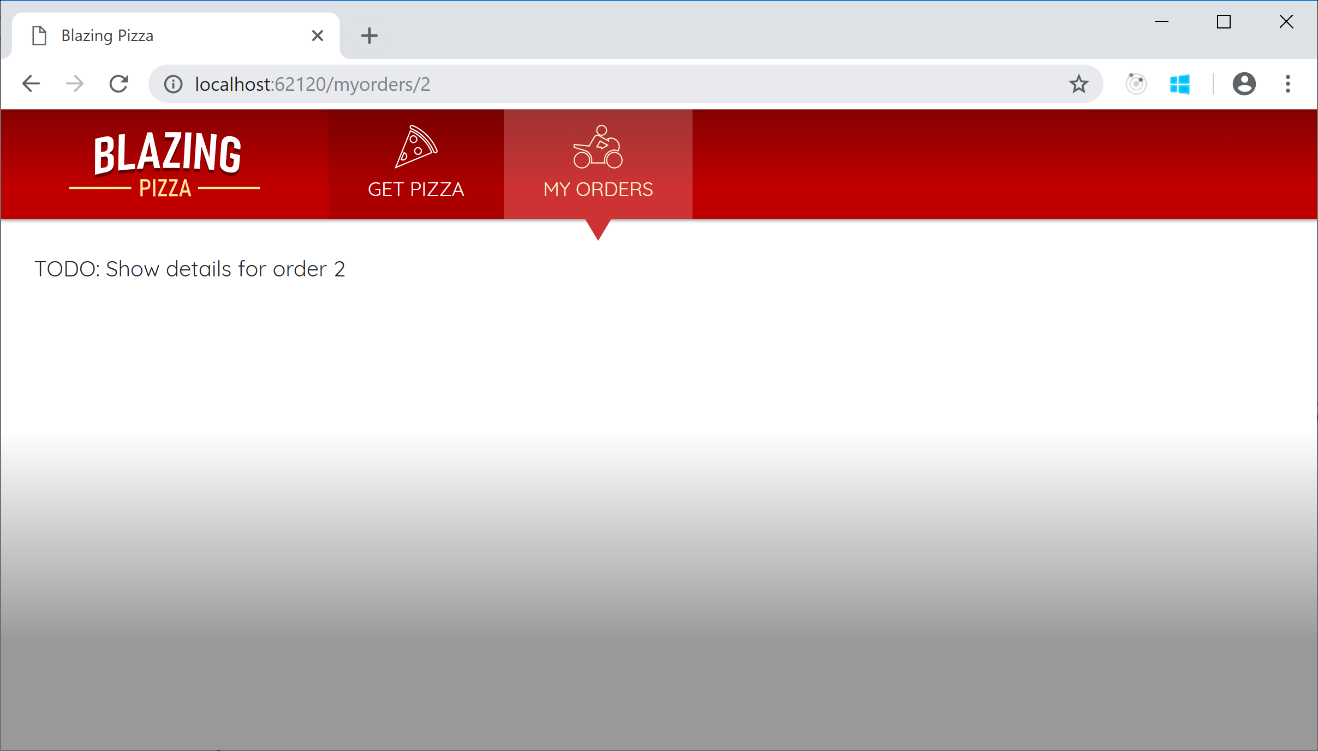
</div>

@code {

[Parameter] public int OrderId { get; set; }

}

This code illustrates how components can receive parameters from the router by declaring them as tokens in the @page directive. If you want to receive a string, the syntax is simply {parameterName}, which matches a [Parameter] name case-insensitively. If you want to receive a numeric value, the syntax is {parameterName:int}, as in the example above. The :int is an example of a route constraint. Other route constraints are supported too.

[](https://user-images.githubusercontent.com/1101362/51805000-cc84bf00-225f-11e9-824b-348561ccc2fa.png)

If you're wondering how routing actually works, let's go through it step-by-step.

* When the app first starts up, the code in Program.cs tells the framework to render App as the root component.
* The App component (in App.razor) contains a <Router>. Router is a built-in component that interacts with the browser's client-side navigation APIs. It registers a navigation event handler that gets notification whenever the user clicks on a link.
* Whenever the user clicks a link, code in Router checks whether the destination URL is within the same SPA (i.e., whether it's under the <base href> value, and it matches some component's declared routes). If it's not, traditional full-page navigation occurs as usual. But if the URL is within the SPA, Router will handle it.
* Router handles it by looking for a component with a compatible @page URL pattern. Each {parameter} token needs to have a value, and the value has to be compatible with any constraints such as :int.
  + If there is a matching component, that's what the Router will render. This is how all the pages in your application have been rendering all along.
  + If there's no matching component, the router tries a full-page load in case it matches something on the server.
  + If the server chooses to re-render the client-side Blazor app (which is also what happens if a visitor is initially arriving at this URL and the server thinks it may be a client-side route), then Blazor concludes the nothing matches on either server or client, so it displays whatever NotFound content is configured.

1. The OrderDetails logic will be quite different from MyOrders. Instead of simply fetching the data once when the component is instantiated, we'll poll the server every few seconds for updated data. This will make it possible to show the order status in (nearly) real-time, and later, to display the delivery driver's location on a moving map. What's more, we'll also account for the possibility of OrderId being invalid. This might happen if:

* No such order exists
* Or later, when we've implemented authentication, if the order is for a different user and you're not allowed to see it

Before we can implement the polling, we'll need to add the following directives at the top of OrderDetails.razor, typically directly under the @page directive:

@using System.Threading

@inject HttpClient HttpClient

You've already seen @inject used with HttpClient, so you know what that is for. Plus, you'll recognize @using from the equivalent in regular .cs files, so this shouldn't be much of a mystery either. Unfortunately, Visual Studio does not yet add @using directives automatically in Razor files, so you do have to write them in yourself when needed.

Now you can implement the polling. Update your @code block as follows:

@code {

[Parameter] public int OrderId { get; set; }

OrderWithStatus orderWithStatus;

bool invalidOrder;

CancellationTokenSource pollingCancellationToken;

protected override void OnParametersSet()

{

// If we were already polling for a different order, stop doing so

pollingCancellationToken?.Cancel();

// Start a new poll loop

PollForUpdates();

}

private async void PollForUpdates()

{

pollingCancellationToken = new CancellationTokenSource();

while (!pollingCancellationToken.IsCancellationRequested)

{

try

{

invalidOrder = false;

orderWithStatus = await HttpClient.GetFromJsonAsync<OrderWithStatus>($"orders/{OrderId}");

}

catch (Exception ex)

{

invalidOrder = true;

pollingCancellationToken.Cancel();

Console.Error.WriteLine(ex);

}

StateHasChanged();

await Task.Delay(4000);

}

}

}

The code is a bit intricate, so be sure to go through it carefully and be sure to understand each aspect of it. Here are some notes:

* This uses OnParametersSet instead of OnInitialized or OnInitializedAsync. OnParametersSet is another component lifecycle method, and it fires when the component is first instantiated *and* any time its parameters change value. If the user clicks a link directly from myorders/2 to myorders/3, the framework will retain the OrderDetails instance and simply update its OrderId parameter in place.
  + As it happens, we haven't provided any links from one "my orders" screen to another, so the scenario never occurs in this application, but it's still the right lifecycle method to use in case we change the navigation rules in the future.
* We're using an async void method to represent the polling. This method runs for arbitrarily long, even while other methods run. async void methods have no way to report exceptions upstream to callers (because typically the callers have already finished), so it's important to use try/catch and do something meaningful with any exceptions that may occur.
* We're using CancellationTokenSource as a way of signalling when the polling should stop. Currently it only stops if there's an exception, but we'll add another stopping condition later.
* We need to call StateHasChanged to tell Blazor that the component's data has (possibly) changed. The framework will then re-render the component. There's no way that the framework could know when to re-render your component otherwise, because it doesn't know about your polling logic.

1. OK, so we are getting the order details, and we're even polling and updating that data every few seconds. But we are still not rendering it in the UI. Let's fix that. Update your <div class="main"> as follows:

<div class="main">

@if (invalidOrder)

{

<h2>Nope</h2>

<p>Sorry, this order could not be loaded.</p>

}

else if (orderWithStatus == null)

{

<text>Loading...</text>

}

else

{

<div class="track-order">

<div class="track-order-title">

<h2>

Order placed @orderWithStatus.Order.CreatedTime.ToLongDateString()

</h2>

<p class="ml-auto mb-0">

Status: <strong>@orderWithStatus.StatusText</strong>

</p>

</div>

<div class="track-order-body">

TODO: show more details

</div>

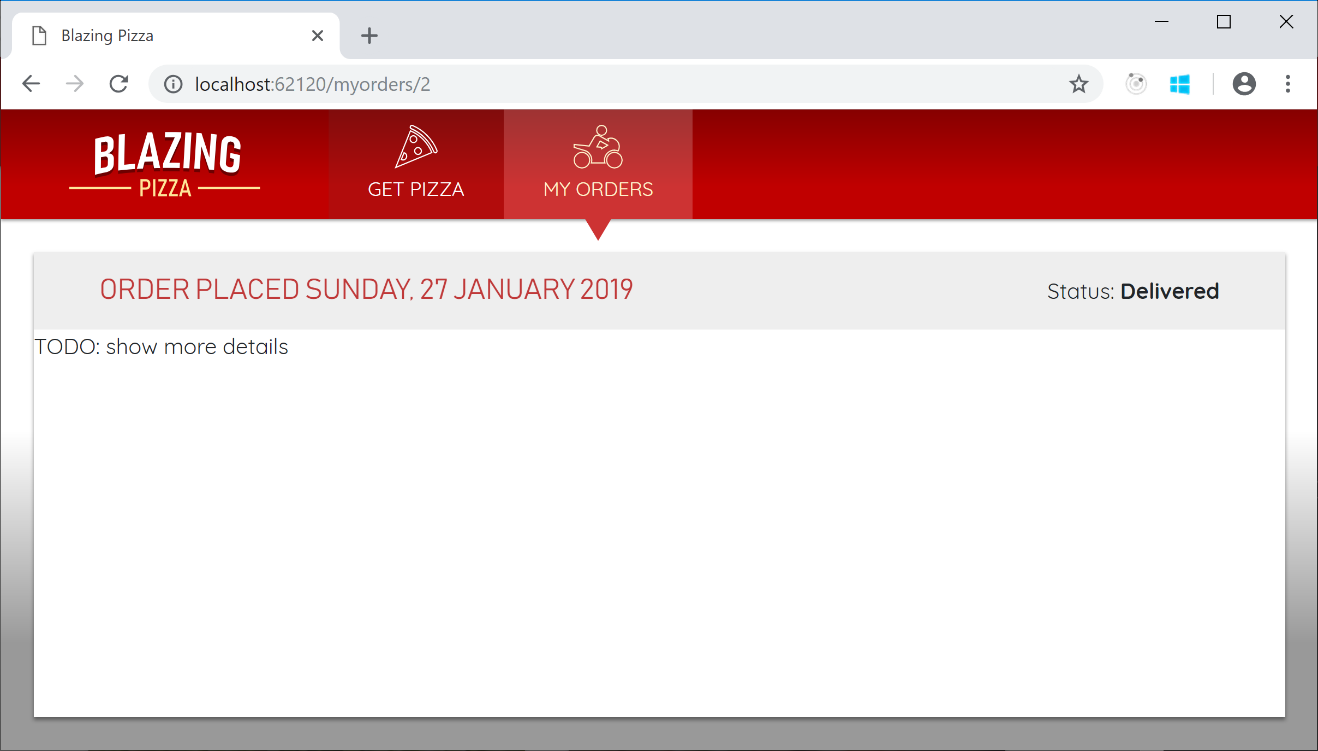
</div>

}

</div>

This accounts for the three main states of the component:

1. If the OrderId value is invalid (i.e., the server reported an error when we tried to retrieve the data)
2. If we have not yet loaded the data
3. If we have some data to show

[](https://user-images.githubusercontent.com/1101362/51805193-5c2b6d00-2262-11e9-98a6-c5a8ec4bb54f.png)

The last bit of UI we want to add is the actual contents of the order. To do this, we will create another reusable component.

1. Create a new file, OrderReview.razor inside the Shared directory, have it receive an Order, then render its contents as follows:

@foreach (var pizza in Order.Pizzas)

{

<p>

<strong>

@(pizza.Size)"

@pizza.Special.Name

(£@pizza.GetFormattedTotalPrice())

</strong>

</p>

<ul>

@foreach (var topping in pizza.Toppings)

{

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

}

</ul>

}

<p>

<strong>

Total price:

£@Order.GetFormattedTotalPrice()

</strong>

</p>

@code {

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

}

1. Finally, back in OrderDetails.razor, replace text TODO: show more details with your new OrderReview component:

<div class="track-order-body">

<div class="track-order-details">

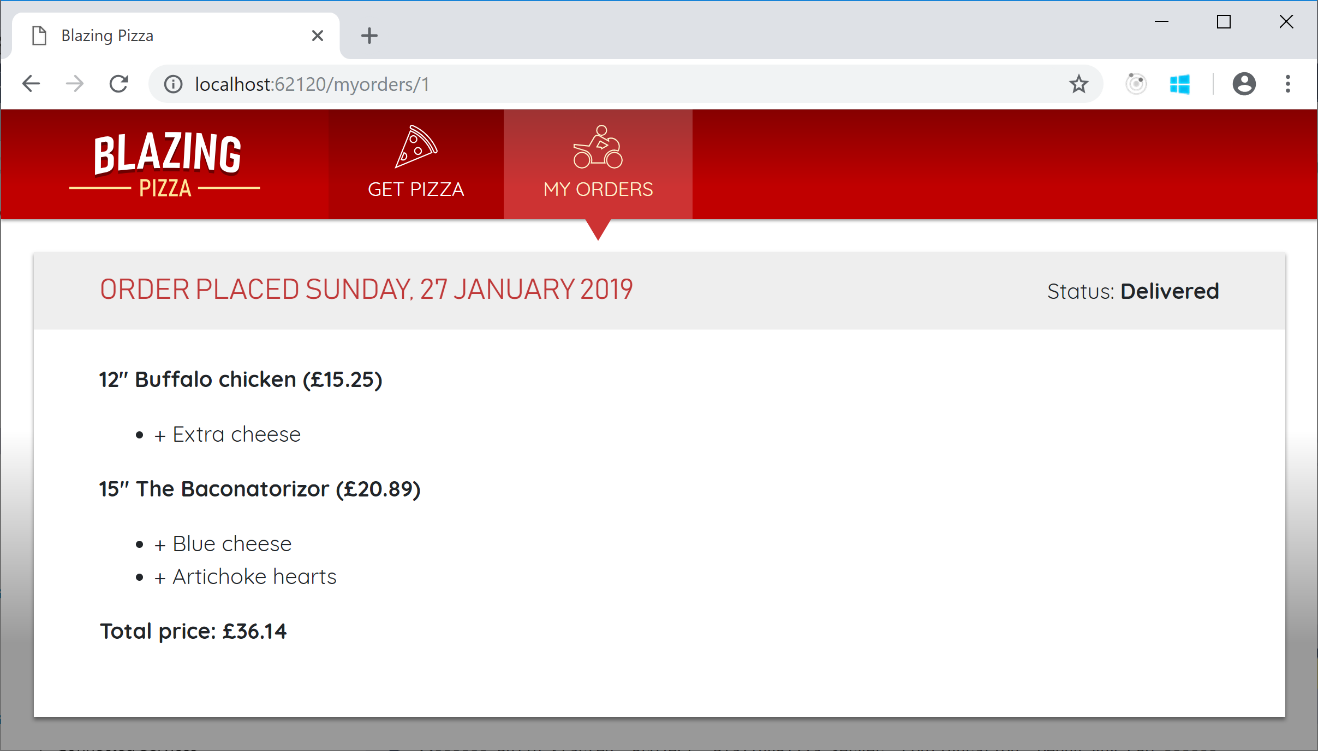
<OrderReview Order="orderWithStatus.Order" />

</div>

</div>

(Don't forget to add the extra div with CSS class track-order-details, as this is necessary for correct styling.)

1. Finally, you have a functional order details display!

[](https://user-images.githubusercontent.com/1101362/51805236-ea9fee80-2262-11e9-814b-8f92f5dbe0de.png)

The backend server will update the order status to simulate an actual dispatch and delivery process. To see this in action, try placing a new order, then immediately view its details.

Initially, the order status will be *Preparing*, then after 10-15 seconds will change to *Out for delivery*, then 60 seconds later will change to *Delivered*. Because OrderDetails polls for updates, the UI will update without the user having to refresh the page.

1. If you deployed your app to production right now, bad things would happen. The OrderDetails logic starts a polling process but doesn't end it. If the user navigated through hundreds of different orders (thereby creating hundreds of different OrderDetails instances), then there would be hundreds of polling processes left running concurrently, even though all, except the last, were pointless because no UI was displaying their results.

You can observe this chaos yourself as follows:

* Navigate to "my orders"
* Click "Track" on any order to get to its details
* Click "Back" to return to "my orders"
* Repeat steps 2 and 3 a lot of times (e.g., 20 times)
* Now, open your browser's debugging tools and look in the network tab. You should see 20 or more HTTP requests being issued every few seconds, because there are 20 or more concurrent polling processes.

This is wasteful of client-side memory and CPU time, network bandwidth, and server resources.

To fix this, we need to make OrderDetails stop the polling once it gets removed from the display. This is simply a matter of using the IDisposable interface.

In OrderDetails.razor, add the following directive at the top of the file, underneath the other directives:

@implements IDisposable

Now if you try to compile the application, the compiler will complain:

error CS0535: 'OrderDetails' does not implement interface member 'IDisposable.Dispose()'

Resolve this by adding the following method inside the @code block:

void IDisposable.Dispose()

{

pollingCancellationToken?.Cancel();

}

The framework calls Dispose automatically when any given component instance is torn down and removed from the UI.

Once you've put in this fix, you can try again to start lots of concurrent polling processes, and you'll see they no longer keep running after the component is gone. Now, the only component that continues to poll is the one that remains on the screen.

1. Right now, once users place an order, the Index component simply resets its state and their order appears to vanish without a trace. This is not very reassuring for users. We know the order is in the database, but users don't know that. It would be nice if, once the order is placed, you navigated to the details display for that order automatically. This is quite easy to do. Switch back to your Index component code. Add the following directive at the top:

@inject NavigationManager NavigationManager

The NavigationManager lets you interact with URIs and navigation state. It has methods to get the current URL, to navigate to a different one, and more.

To use this, update the PlaceOrder code so it calls NavigationManager.NavigateTo:

async Task PlaceOrder()

{

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

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

order = new Order();

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

}

Now as soon as the server accepts the order, the browser will switch to the "order details" display and begin polling for updates.