Blazor

Module 7: Forms and Validation

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 7: Forms and Validation 1](#_Toc38805815)

[Exercise 1: Validation 3](#_Toc38805816)

# Lab 7: Forms and Validation

#### 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:

* Introduce Server-Side Validation
* Introduce Client-Side Validation

#### 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: Validation

#### Objectives

In this exercise, you will:

* Introduce Server-Side Validation
* Introduce Client-Side Validation

#### Scenario

If you look at the Order class in *BlazingPizza.Shared*, you might notice that it holds a DeliveryAddress property of type Address. However, nothing in the pizza ordering flow populates this data yet, so all your orders just have a blank delivery address. It's time to fix this by adding a "checkout" screen that requires customers to enter a valid address.

Task 1: Inserting checkout into the flow

1. Start by opening the solution file **BlazingPizza.sln** located under **\Labs\Module 07 -Validation\Begin**.
2. Add a new page component, Checkout.razor, with a @page directive matching the URL /checkout. For the initial markup, let's display the details of the order using your OrderReview component:

<div class="main">

<div class="checkout-cols">

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

<h4>Review order</h4>

<OrderReview Order="OrderState.Order" />

</div>

</div>

<button class="checkout-button btn btn-warning" @onclick="PlaceOrder">

Place order

</button>

</div>

1. To implement PlaceOrder, copy the method with that name from Index.razor into Checkout.razor:

@code {

async Task PlaceOrder()

{

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

OrderState.ResetOrder();

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

}

}

1. As usual, you will need to @inject values for OrderState, HttpClient, and NavigationManager so that it can compile, just like you did in Index.razor.

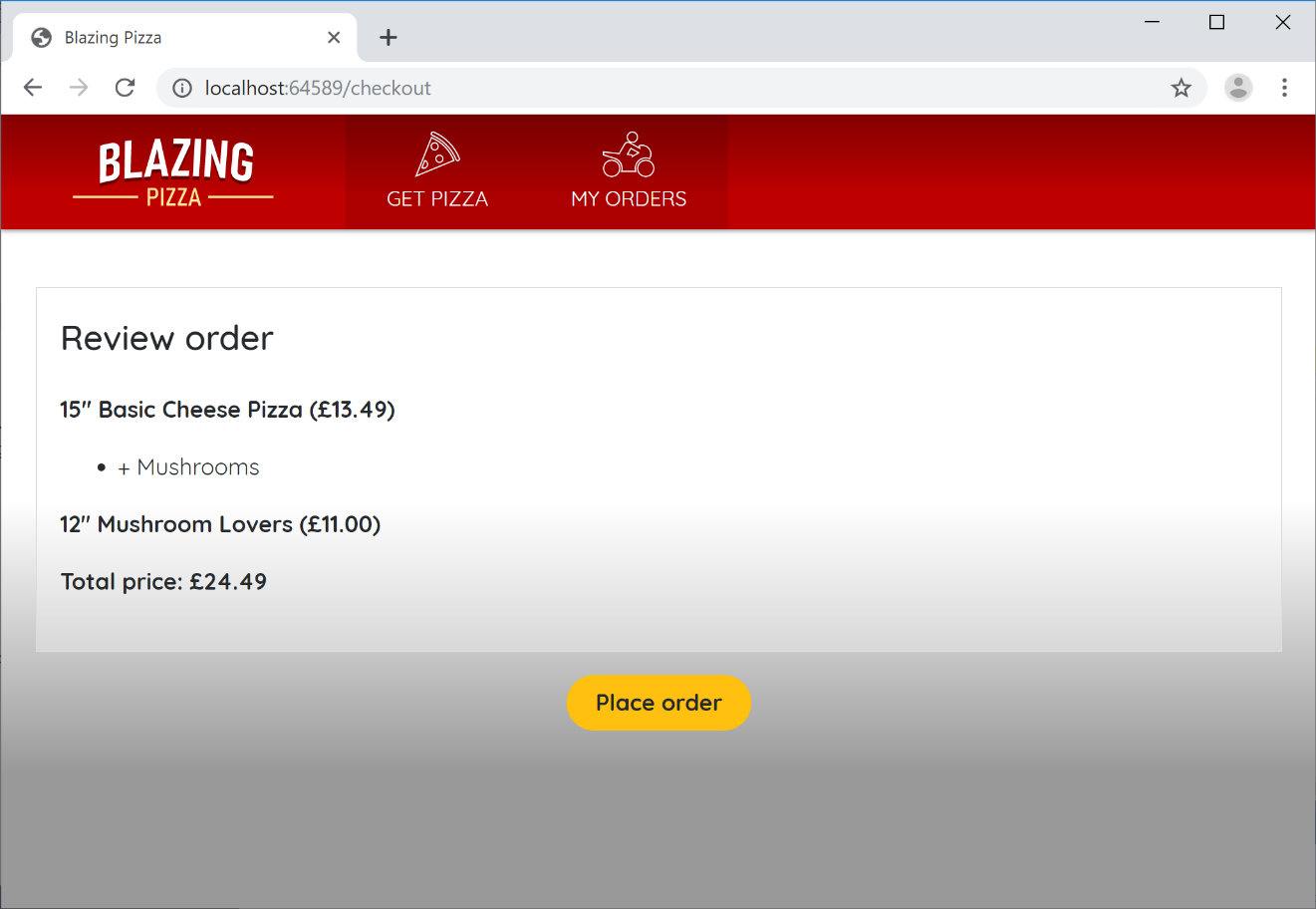
Next, let's bring customers here when they try to submit orders. Back in Index.razor, make sure you've deleted the PlaceOrder method, and then change the order submission button into a regular HTML link to the /checkout URL, i.e.:

<a href="checkout" class="btn btn-warning" disabled="@(OrderState.Order.Pizzas.Count == 0)">

Order >

</a>

1. Now, when you run the app, you should be able to reach the checkout page by clicking the Order button, and from there can click Place order to confirm it.

[](https://user-images.githubusercontent.com/1101362/59218134-674ebc00-8bb7-11e9-97d6-0c9985f10acf.png)

1. We now have a good place to put some UI for entering a delivery address. As usual, let's factor this into a reusable component. You never know when you're going to be asking for addresses in other places.

Create a new component in the BlazingPizza.Client project's Shared folder called AddressEditor.razor. It's going to be a general way to edit Address instances, so have it receive a parameter of this type:

@code {

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

}

1. The markup here is going to be a bit tedious, so you will probably want to copy and paste it. We'll need input elements for each of the properties on an Address:

<div class="form-field">

<label>Name:</label>

<div>

<input @bind="Address.Name" />

</div>

</div>

<div class="form-field">

<label>Line 1:</label>

<div>

<input @bind="Address.Line1" />

</div>

</div>

<div class="form-field">

<label>Line 2:</label>

<div>

<input @bind="Address.Line2" />

</div>

</div>

<div class="form-field">

<label>City:</label>

<div>

<input @bind="Address.City" />

</div>

</div>

<div class="form-field">

<label>Region:</label>

<div>

<input @bind="Address.Region" />

</div>

</div>

<div class="form-field">

<label>Postal code:</label>

<div>

<input @bind="Address.PostalCode" />

</div>

</div>

@code {

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

}

1. Finally, you can use your AddressEditor inside the Checkout.razor component:

<div class="checkout-cols">

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

... leave this div unchanged ...

</div>

<div class="checkout-delivery-address">

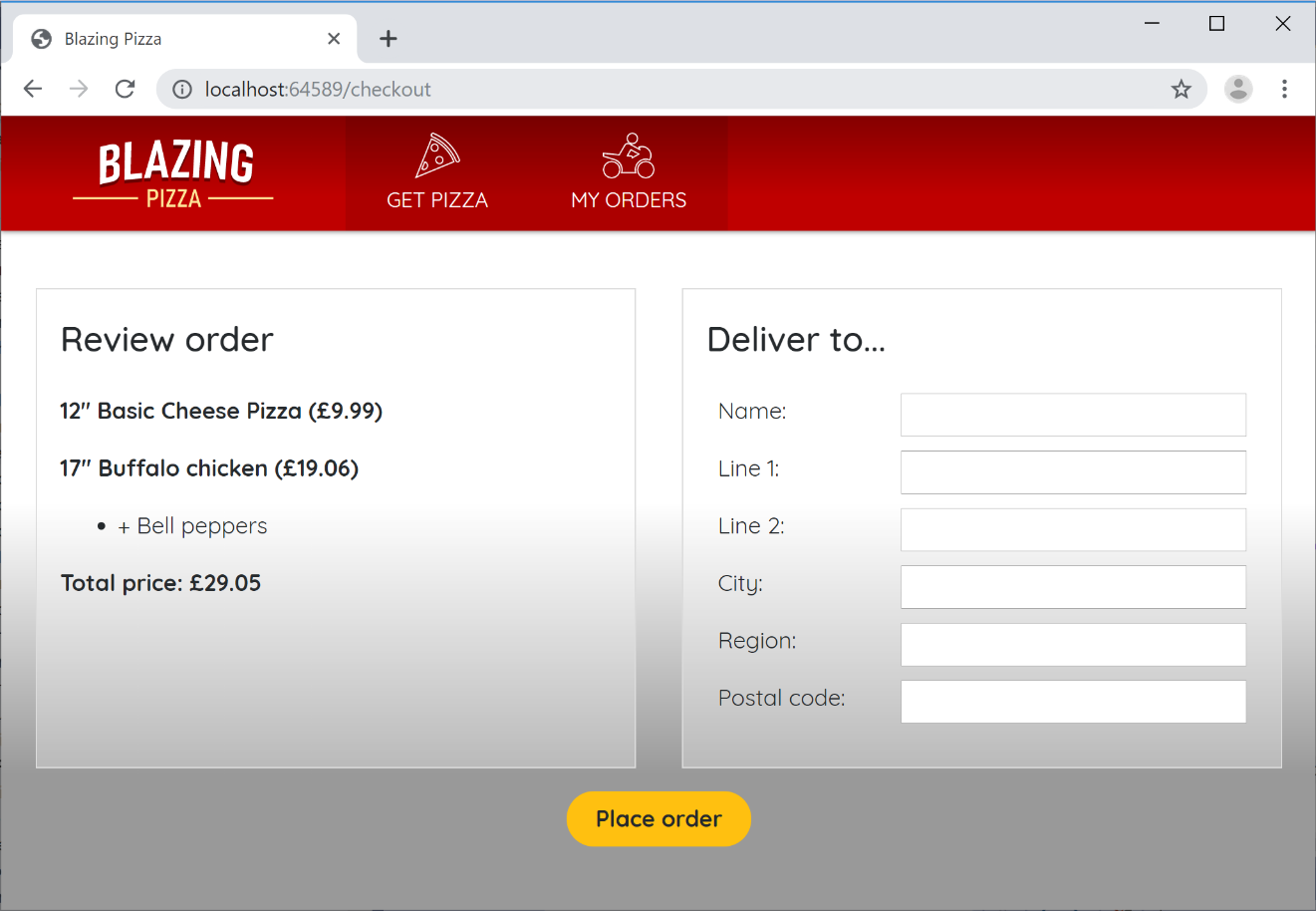
<h4>Deliver to...</h4>

<AddressEditor Address="OrderState.Order.DeliveryAddress" />

</div>

</div>

1. Your checkout screen now asks for a delivery address:

[](https://user-images.githubusercontent.com/1101362/59219467-76833900-8bba-11e9-960b-67aec2e2f8c7.png)

If you submit an order now, any address data that you entered will actually be saved in the database with the order, because it's all part of the Order object that gets serialized and sent to the server.

If you're really keen to verify the data gets saved, consider downloading a tool such as [DB Browser for SQLite](https://sqlitebrowser.org/) to inspect the contents of your pizza.db file. But you don't strictly need to do this.

Alternatively, you can set a breakpoint inside BlazingPizza.Server's OrderController.PlaceOrder method, and use the debugger to inspect the incoming Order object. Here you should be able to see the backend server receive the address data you typed in.

Task 2: Adding Server-side Validation

1. Unfortunately, customers can still leave the "delivery address" fields blank and merrily order a pizza to be delivered nowhere in particular. When it comes to validation, it's normal to implement rules both on the server and on the client:

* Client-side validation is a courtesy to your users. It can provide instant feedback while they are editing a form. However, it can easily be bypassed by anyone with a basic knowledge of the browser dev tools.
* Server-side validation is where the real enforcement is.

As such, it's usually best to start by implementing server-side validation, so you know your app is robust no matter what happens client-side. If you go and look at OrdersController.cs in the *BlazingPizza.Server* project, you will see that this API endpoint is decorated with the [ApiController] attribute:

[Route("orders")]

[ApiController]

public class OrdersController : Controller

{

// ...

}

[ApiController] adds various server-side conventions, including enforcement of DataAnnotations validation rules. So all we need to do is put some DataAnnotations validation rules onto the model classes.

1. Open Address.cs from the *BlazingPizza.Shared* project and put a [Required] attribute onto each of the properties except for Id (which is autogenerated, because it is the primary key) and Line2, since not all addresses need a second line. You can also place some [MaxLength] attributes if you wish, or any other DataAnnotations rules:

using System.ComponentModel.DataAnnotations;

namespace BlazingPizza

{

public class Address

{

public int Id { get; set; }

[Required, MaxLength(100)]

public string Name { get; set; }

[Required, MaxLength(100)]

public string Line1 { get; set; }

[MaxLength(100)]

public string Line2 { get; set; }

[Required, MaxLength(50)]

public string City { get; set; }

[Required, MaxLength(20)]

public string Region { get; set; }

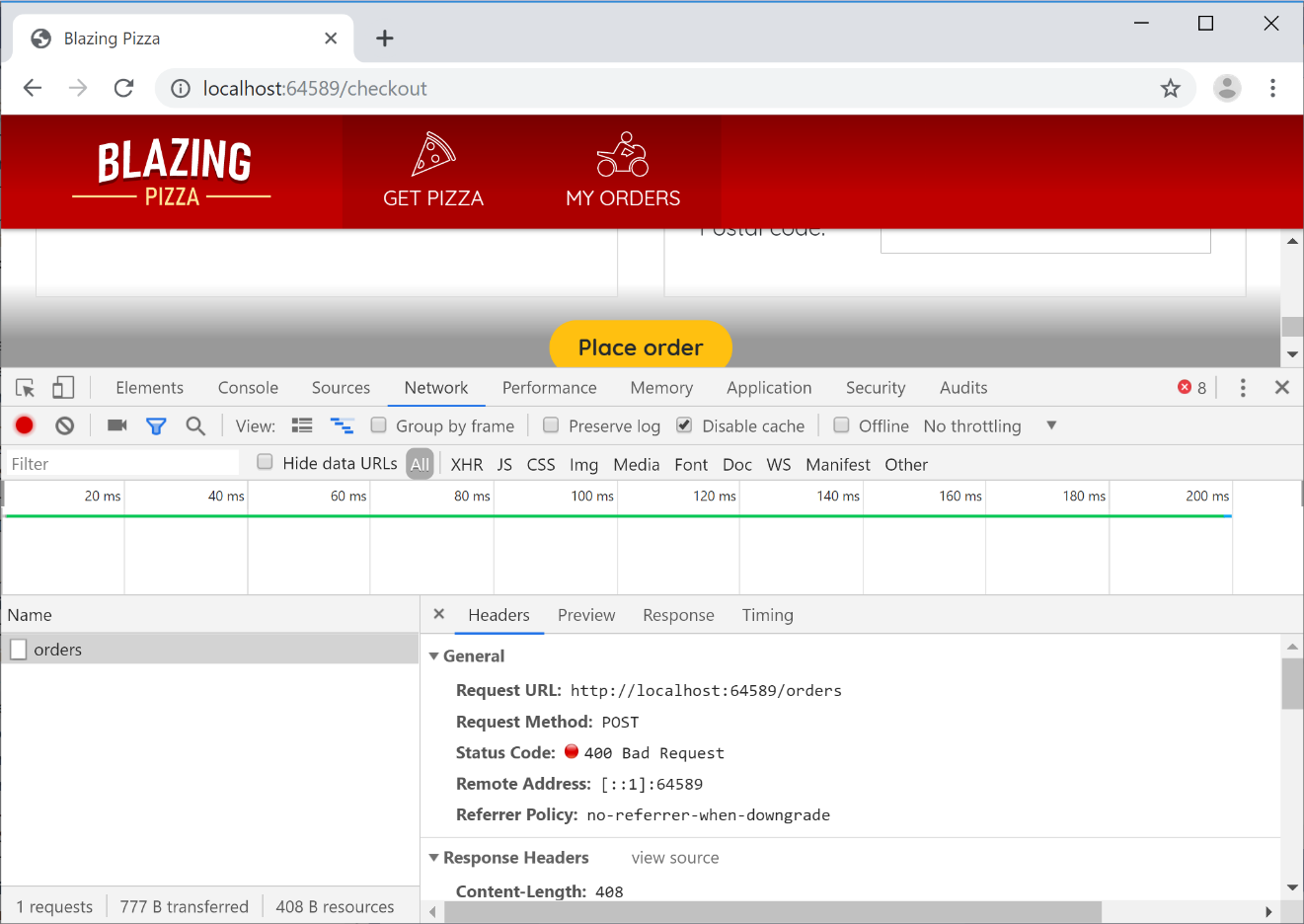
[Required, MaxLength(20)]

public string PostalCode { get; set; }

}

}

1. Now, recompile and run your application, and you should be able to observe the validation rules being enforced on the server. If you try to submit an order with a blank delivery address, then the server will reject the request and you'll see an HTTP 400 ("Bad Request") error in the browser's *Network* tab:

[](https://user-images.githubusercontent.com/1101362/59220316-93207080-8bbc-11e9-8bbb-f3f5ec8a29d6.png)

Task 3: Adding Client-side Validation

1. Blazor has a comprehensive system for data entry forms and validation. We'll now use this to apply the same DataAnnotations rules on the client that are already being enforced on the server.

The way Blazor's forms and validation system works is based around something called an EditContext. An EditContext tracks the state of an editing process, so it knows which fields have been modified, what data has been entered, and whether or not the fields are valid. Various built-in UI components hook into the EditContext both to read its state (e.g., display validation messages) and to write to its state (e.g., to populate it with the data entered by the user).

1. One of the most important built-in UI components for data entry is the EditForm. This renders as an HTML <form> tag, but also sets up an EditContext to track what's going on inside the form. To use this, go to your *Checkout.razor* component, and wrap an EditContext around the whole of the contents of the main div:

<div class="main">

<EditForm Model="OrderState.Order.DeliveryAddress">

<div class="checkout-cols">

... leave unchanged ...

</div>

<button class="checkout-button btn btn-warning" @onclick="PlaceOrder">

Place order

</button>

</EditForm>

</div>

You can have multiple EditForm components at once, but they can't overlap (because HTML's <form> elements can't overlap). By specifying a Model, we're telling the internal EditContext which object it should validate when the form is submitted (in this case, the delivery address).

1. Let's start by displaying validation messages in a very basic (and not very attractive) way. Inside the EditForm, right at the bottom, add the following two components:

<DataAnnotationsValidator />

<ValidationSummary />

The DataAnnotationsValidator hooks into events on the EditContext and executes DataAnnotations rules. If you wanted to use a different validation system other than DataAnnotations, you'd swap DataAnnotationsValidator for something else.

The ValidationSummary simply renders an HTML <ul> containing any validation messages from the EditContext.

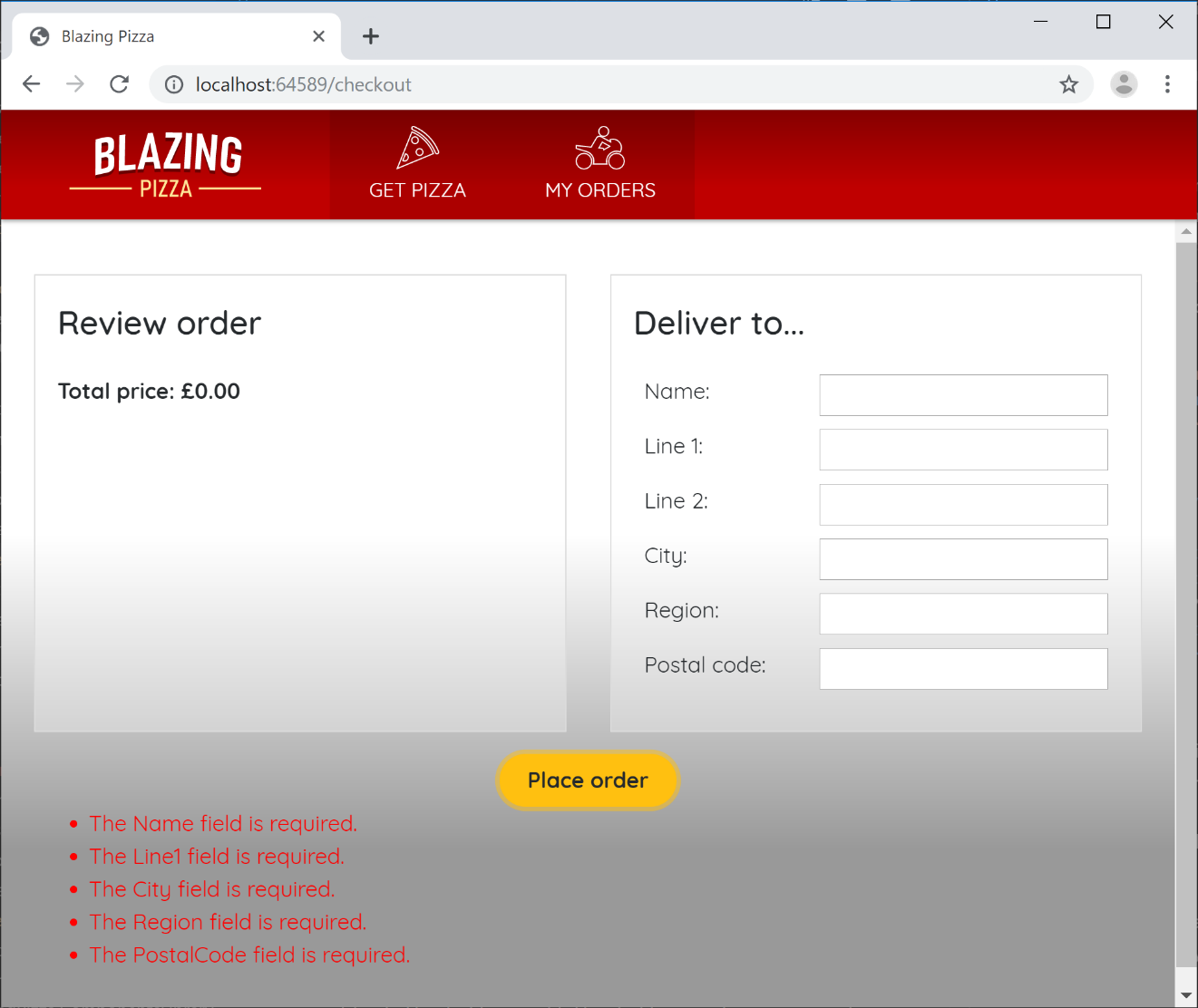
1. If you ran your application now, you could still submit a blank form (and the server would still respond with an HTTP 400 error). That's because your <button> isn't actually a submit button. Modify the button by adding type="submit" and **removing** its @onclick attribute entirely.

Next, instead of triggering PlaceOrder directly from the button, you need to trigger it from the EditForm. Add the following OnValidSubmit attribute onto the EditForm:

<EditForm Model="OrderState.Order.DeliveryAddress" OnValidSubmit="PlaceOrder">

As you can probably guess, the <button> no longer triggers PlaceOrder directly. Instead, the button just asks the form to be submitted. And then the form decides whether it's valid, and if it is, then it will call PlaceOrder.

1. Try it out: you should no longer be able to submit an invalid form, and you'll see validation messages (albeit unattractive ones).

[](https://user-images.githubusercontent.com/1101362/59221577-85201f00-8bbf-11e9-9de6-f24f93a6a483.png)

1. Obviously, it's not clean to display all the validation messages so far away from the textboxes. Let's move them to better places.

Start by removing the <ValidationSummary> component entirely. Then, switch over to AddressEditor.razor, and add separate <ValidationMessage> components next to each of the form fields. For example,

<div class="form-field">

<label>Name:</label>

<div>

<input @bind="Address.Name" />

<ValidationMessage For="() => Address.Name" />

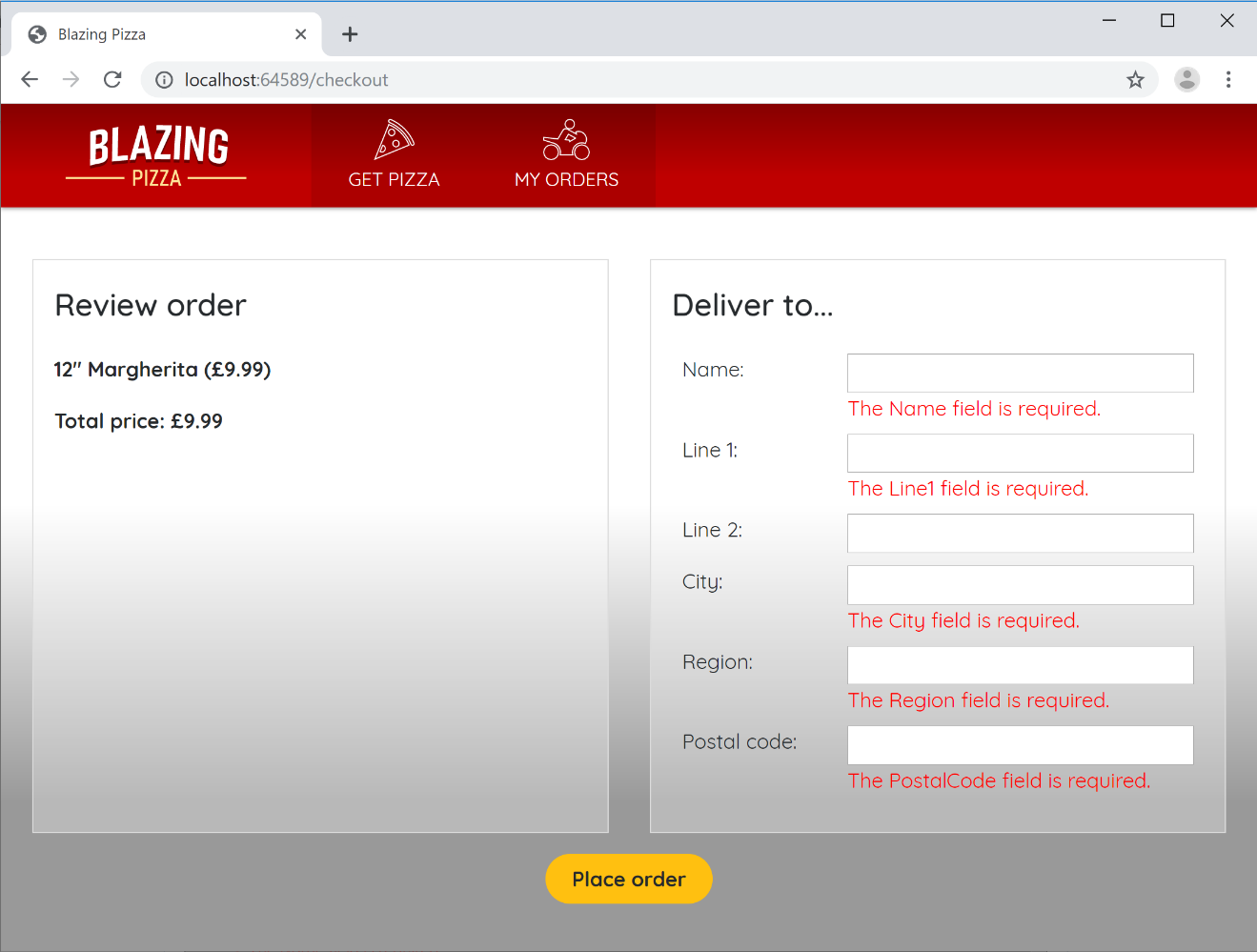
</div>

</div>

1. Do the equivalent for all the form fields.

In case you're wondering, the syntax @(() => Address.Name) is a *lambda expression*, and we use this syntax as a way of describing which property to read the metadata from, without actually evaluating the property's value.

Now things look a lot better:

[](https://user-images.githubusercontent.com/1101362/59221927-4b034d00-8bc0-11e9-96ef-6c41ad727cb7.png)

1. If you want, you can improve the readability of the messages by specifying custom ones. For example, instead of displaying The City field is required, you could go to Address.cs and do this:

[Required(ErrorMessage = "How do you expect to receive the pizza if we don't even know what city you're in?"), MaxLength(50)]

public string City { get; set; }

Task 4: Better Validation UX Using The Built-In Input Components

1. The user experience is still not great, because once the validation messages are displayed, they remain on the screen until you click *Place order* again, even if you have edited the field values. Try it out and see how it feels pretty basic!

To improve on this, you can replace the low-level HTML input elements with Blazor's built-in input components. They know how to hook more deeply into the EditContext:

* When they are edited, they notify the EditContext immediately so it can refresh validation status.
* They also receive notifications about validity from the EditContext, so they can highlight themselves as either valid or invalid as they user edits them.

1. Go back to *AddressEditor.razor* once again. Replace each of the <input> elements with a corresponding <InputText>. For example,

<div class="form-field">

<label>Name:</label>

<div>

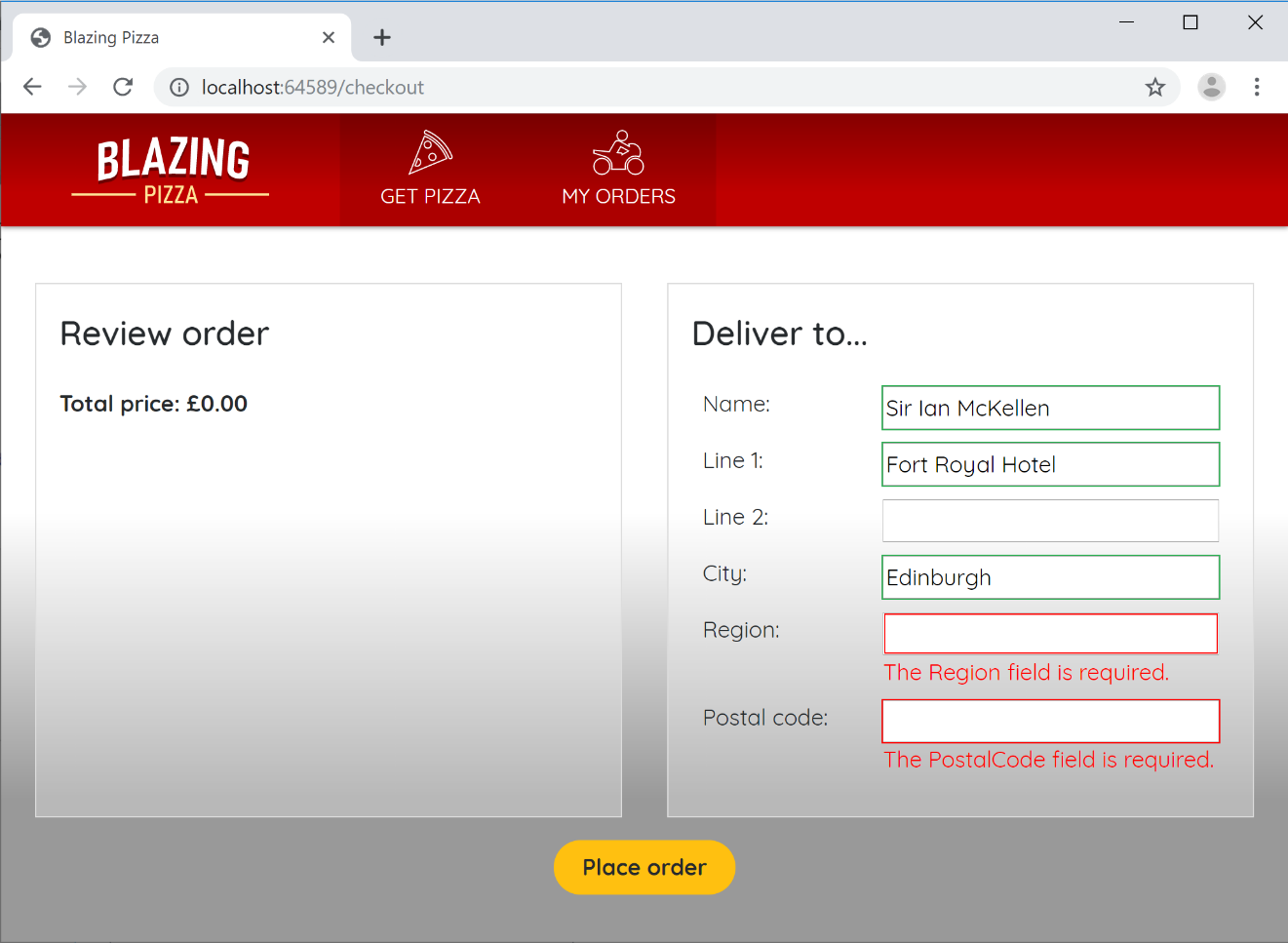
<InputText @bind-Value="Address.Name" />

<ValidationMessage For="() => Address.Name" />

</div>

</div>

Do this for all the properties. The behavior is now much better! As well as having the validation messages update individually for each form field as you change focus, you'll get a neat "valid" or "invalid" highlight around each one:

[](https://user-images.githubusercontent.com/1101362/59222864-6a9b7500-8bc2-11e9-9f90-ae1d47fc23d7.png)

The green/red styling is achieved by applying CSS classes, so you can change the appearance of these effects or remove them entirely if you wish.

InputText isn't the only built-in input component, though it is the only one we need in this case. Others include InputCheckbox, InputDate, InputSelect, and more.