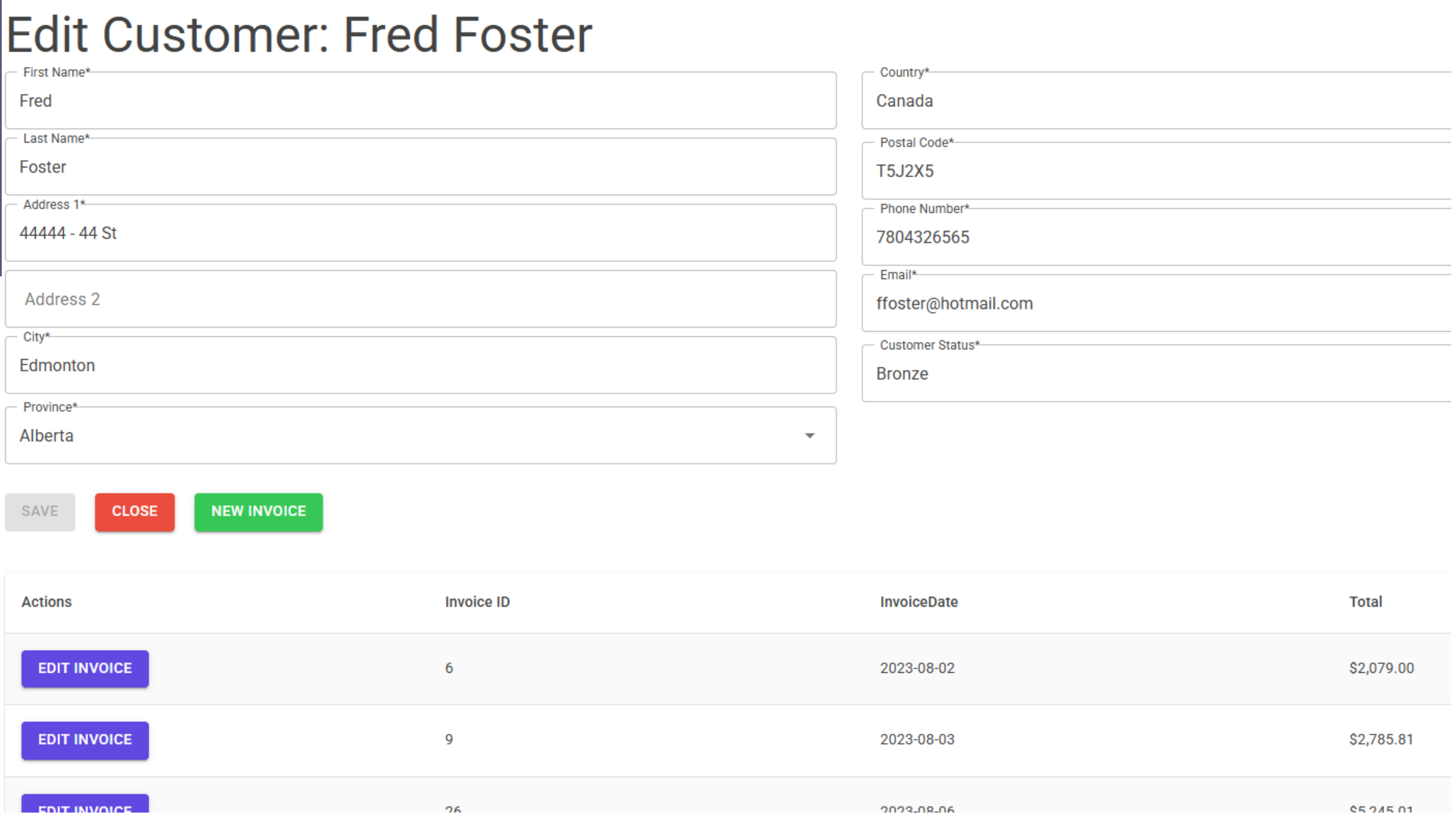
Session 6 - Single Table CRUD - No Delete Using MudBlazor

Overview

- NOTE: We are reusing the code from the OLTP – CustomerService.linq
- Copy the “Customer Edit” view model (CustomerEditView) to the view model folder.
- Add the “GetCustomer()” method to the Customer Service class.
- Create a Blazor Page for adding or editing customers based on the customer ID.
- Updated the “New” and “Edit” methods within the “Customer Search” page.
- Add page validation based on Add/Edit rules.
- Copy the “AddEditCustomer()” method to the “Customer Service.”

Overview



Edit Customer: Fred Foster

First Name*: Fred

Last Name*: Foster

Address 1*: 44444 - 44 St

Address 2

City*: Edmonton

Province*: Alberta

Country*: Canada

Postal Code*: T5J2X5

Phone Number*: 7804326565

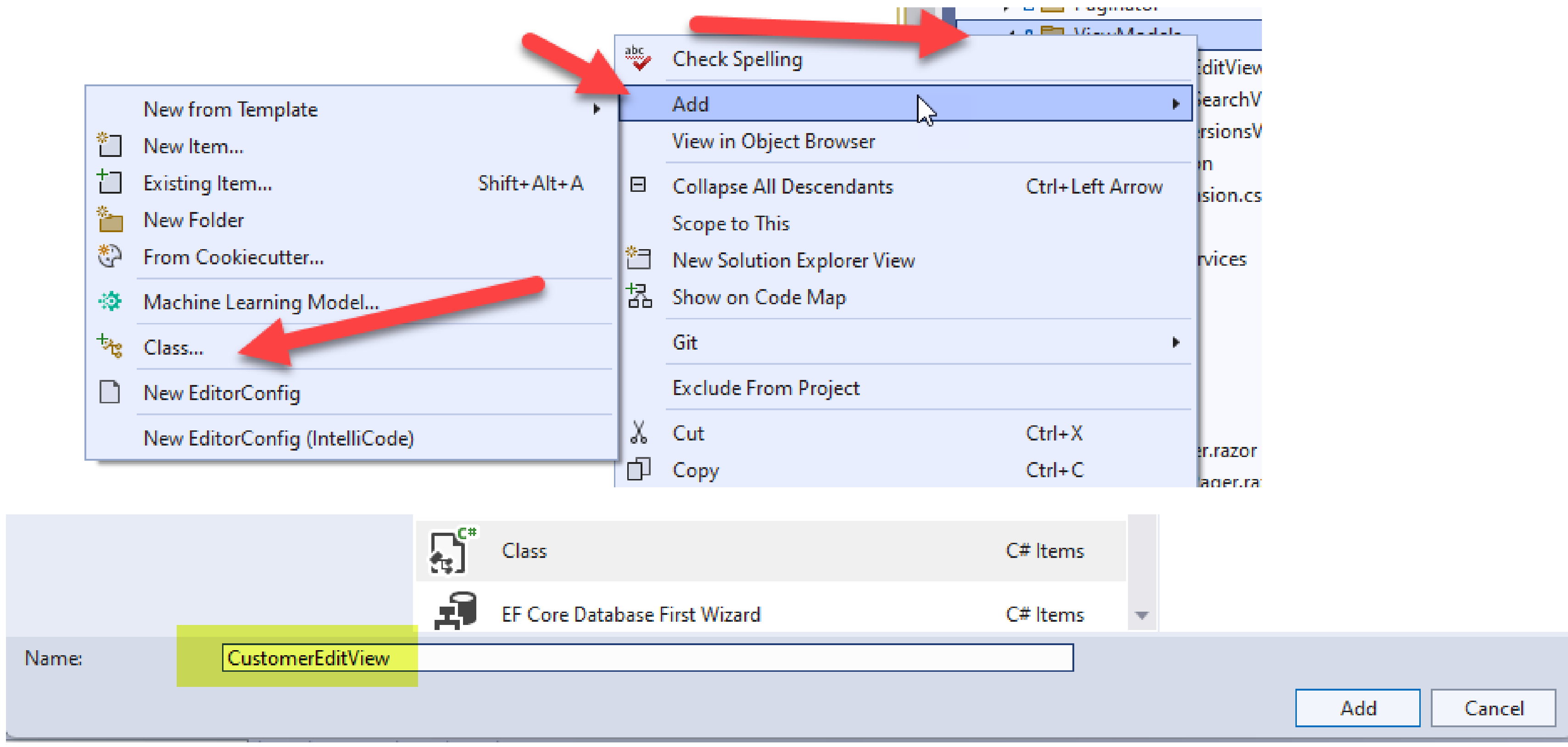
Email*: ffoster@hotmail.com

Customer Status*: Bronze

Actions	Invoice ID	InvoiceDate	Total
EDIT INVOICE	6	2023-08-02	$2,079.00
EDIT INVOICE	9	2023-08-03	$2,785.81
EDIT INVOICE	26	2023-08-06	$5,245.01

Creating Customer Edit View Model

- We must create a view model for adding/editing the “Customer” table by creating a new class called CustomerEditView to the ViewModels folder.

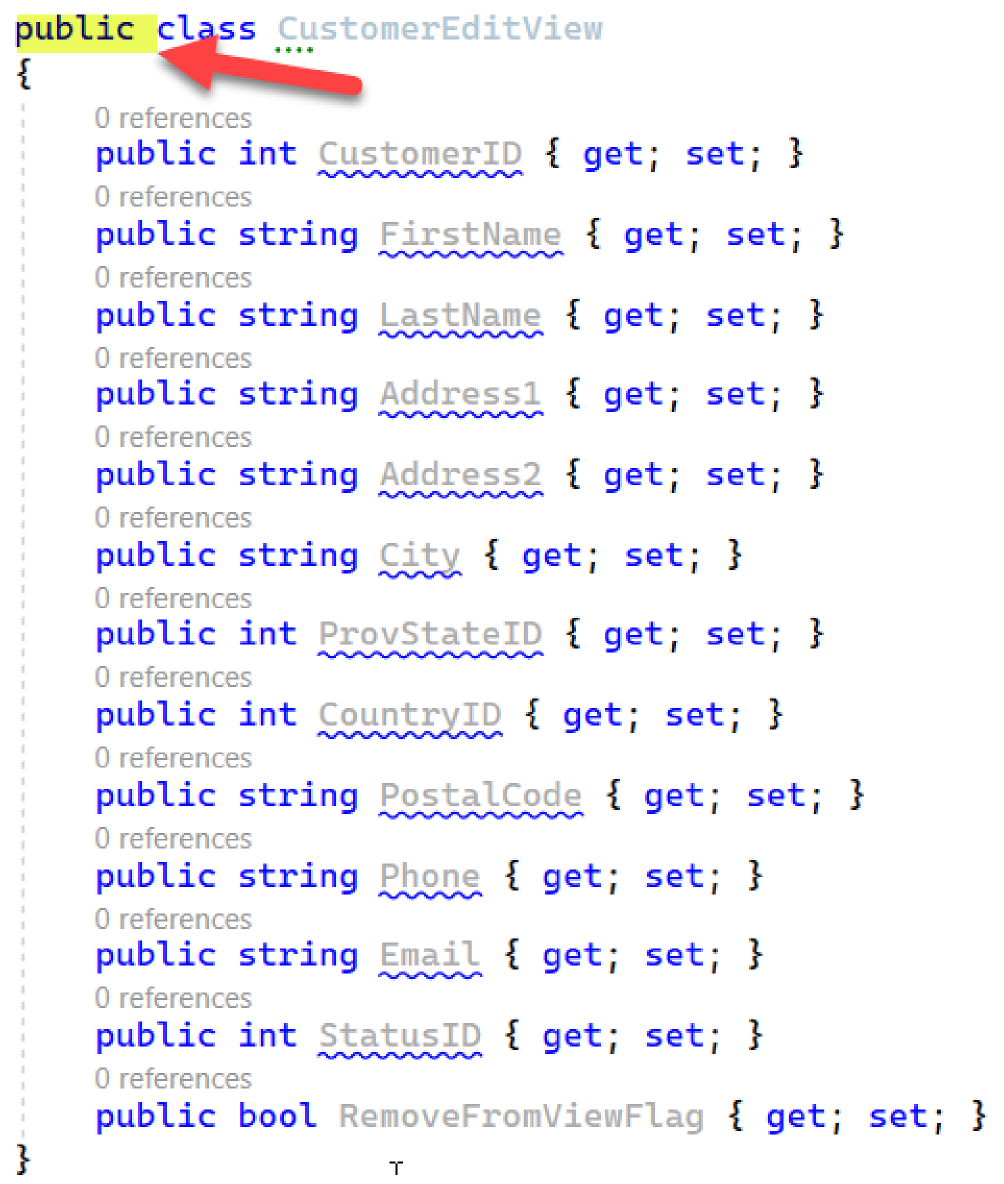


Name: CustomerEditView

Creating Customer Edit View Model (Continue)

- Copy the view model from the "CustomerService.linq"

CustomerEditView.cs

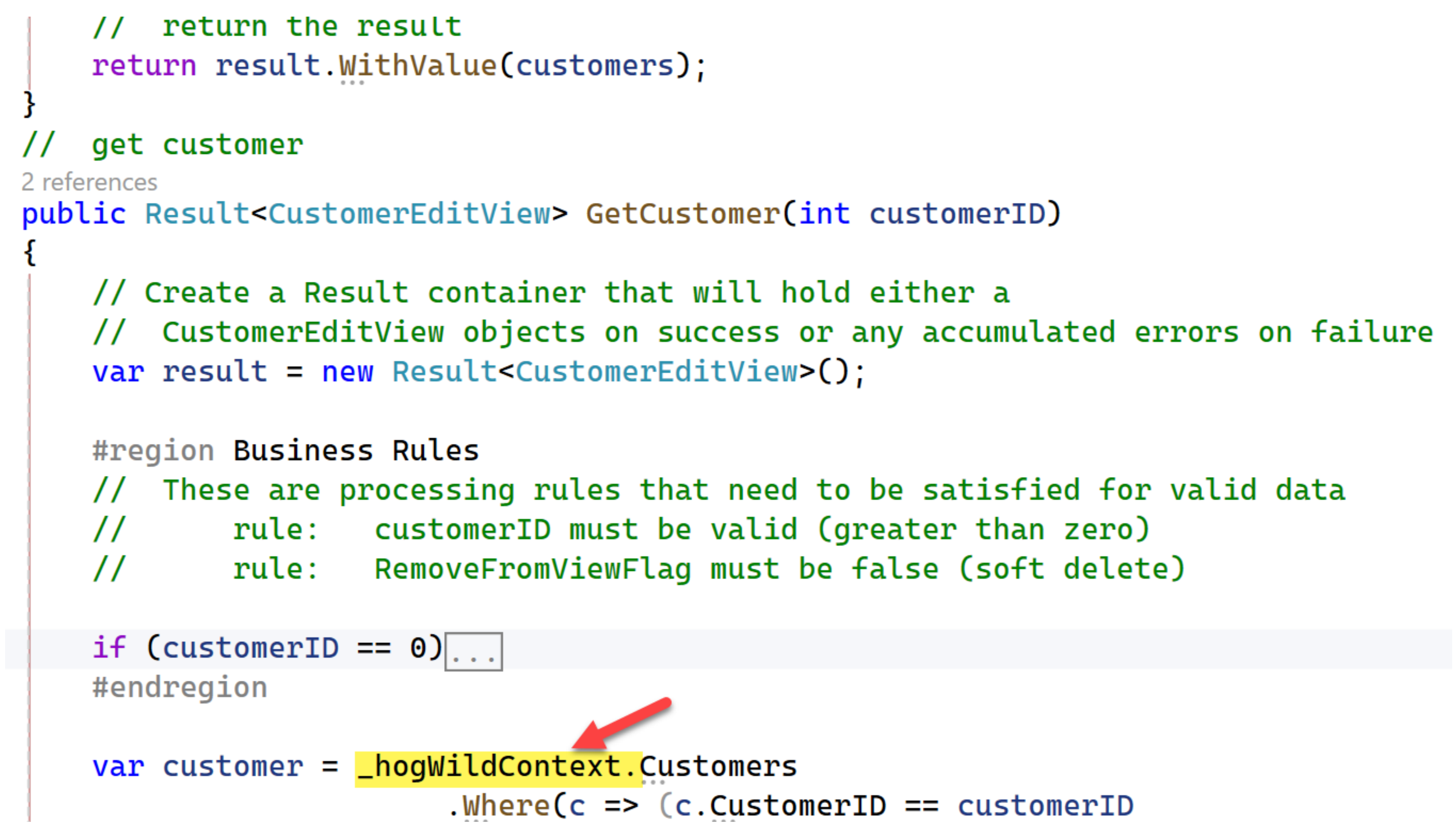


```
public class CustomerEditView
{
        0 references
        public int CustomerID { get; set; }
        0 references
        public string FirstName { get; set; }
        0 references
        public string LastName { get; set; }
        0 references
        public string Address1 { get; set; }
        0 references
        public string Address2 { get; set; }
        0 references
        public string City { get; set; }
        0 references
        public int ProvStateID { get; set; }
        0 references
        public int CountryID { get; set; }
        0 references
        public string PostalCode { get; set; }
        0 references
        public string Phone { get; set; }
        0 references
        public string Email { get; set; }
        0 references
        public int StatusID { get; set; }
        0 references
        public bool RemoveFromViewFlag { get; set; }
}
```

Update Customer Service with "GetCustomer()" Method

CustomerService.cs

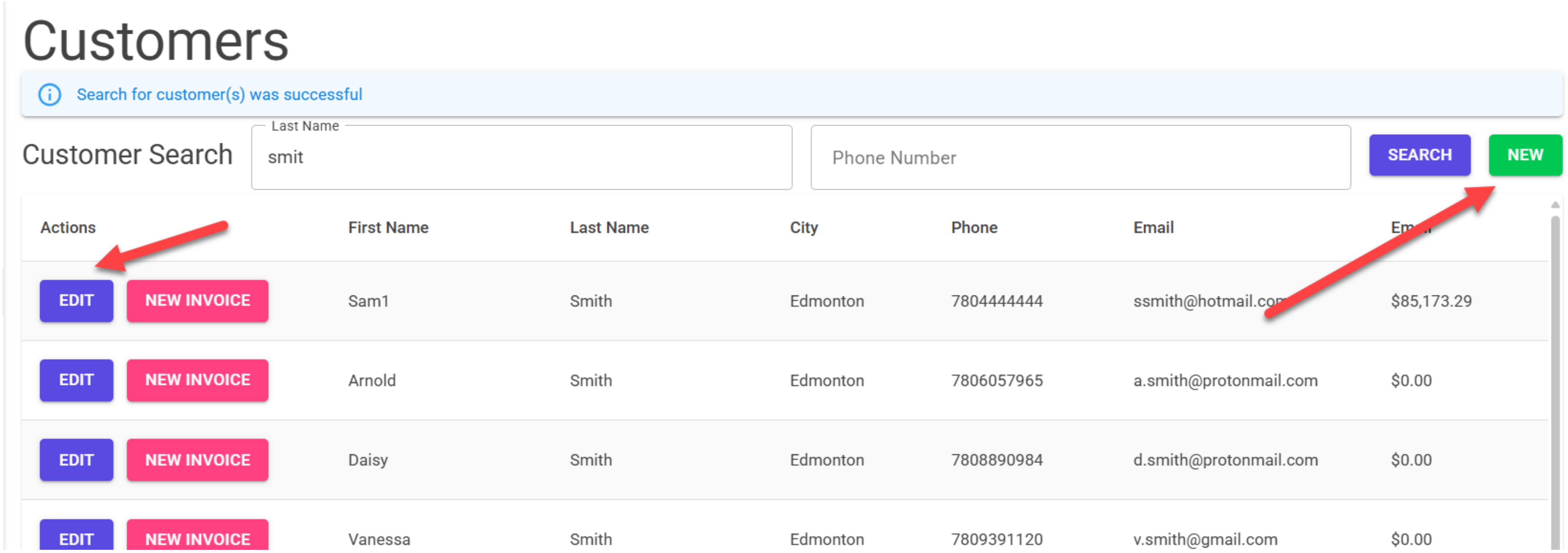
- Copy your method into the "Customer Service" class.
- Update the return to use the HogWildContext.
- **NOTE: If you have followed the BYSResults pattern that we have created by building the functionality in LINQPad, then you do not have to do this step as we are already using the "_hogWildContext"**



```
        //  return the result
        return result.WithValue(customers);
    }
    //  get customer
    2 references
    public Result<CustomerEditView> GetCustomer(int customerID)
    {
        // Create a Result container that will hold either a
        //  CustomerEditView objects on success or any accumulated errors on failure
        var result = new Result<CustomerEditView>();

        #region Business Rules
        //  These are processing rules that need to be satisfied for valid data
        //      rule:   customerID must be valid (greater than zero)
        //      rule:   RemoveFromViewFlag must be false (soft delete)

        if (customerID == 0)...
        #endregion

        var customer = _hogWildContext.Customers
                            .Where(c => (c.CustomerID == customerID
```

Calling Customer Edit

- The customer edit will be called from the customer list screen. When a user either click the “New” or “Edit” button, the customer edit screen will be loaded.



Customers

Search for customer(s) was successful

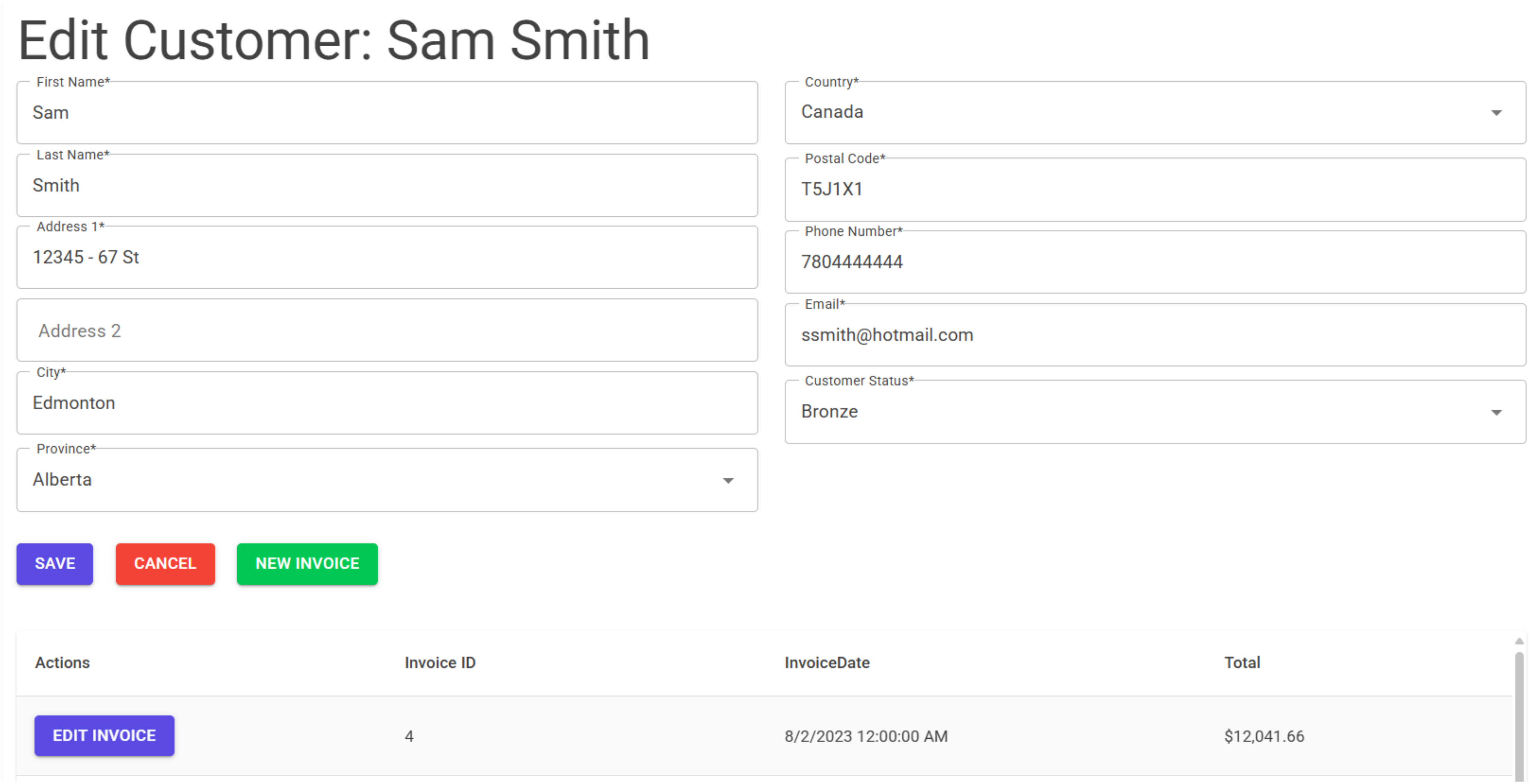
Customer Search

Last Name: smit

Phone Number

Actions	First Name	Last Name	City	Phone	Email	Em
EDIT NEW INVOICE	Sam1	Smith	Edmonton	7804444444	ssmith@hotmail.com	$85,173.29
EDIT NEW INVOICE	Arnold	Smith	Edmonton	7806057965	a.smith@protonmail.com	$0.00
EDIT NEW INVOICE	Daisy	Smith	Edmonton	7808890984	d.smith@protonmail.com	$0.00
EDIT NEW INVOICE	Vanessa	Smith	Edmonton	7809391120	v.smith@gmail.com	$0.00

Customer Edit



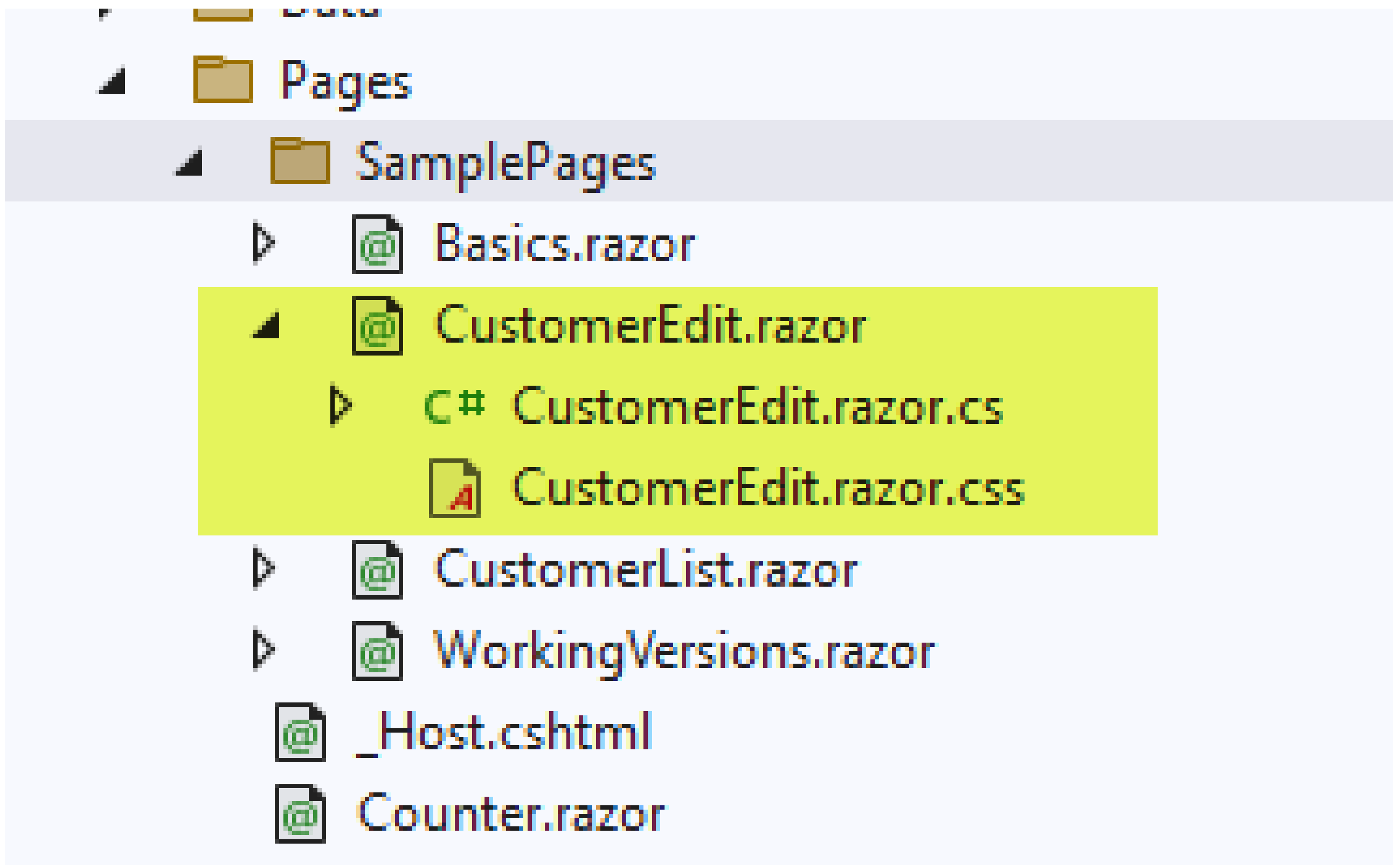
Edit Customer: Sam Smith

First Name*	Sam
Last Name*	Smith
Address 1*	12345 - 67 St
Address 2	
City*	Edmonton
Province*	Alberta
Country*	Canada
Postal Code*	T5J1X1
Phone Number*	7804444444
Email*	ssmith@hotmail.com
Customer Status*	Bronze

Actions	Invoice ID	InvoiceDate	Total
EDIT INVOICE	4	8/2/2023 12:00:00 AM	$12,041.66

Customer Edit Page

- In the “Sample Page” folder, add three new files
  - CustomerEdit.razor (Razor Component)
  - CustomerEdit.razor.cs (Class)
  - CustomerEdit.razor.css (Text File)


Pages
SamplePages
Basics.razor
CustomerEdit.razor
CustomerEdit.razor.cs
CustomerEdit.razor.css
CustomerList.razor
WorkingVersions.razor
_Host.cshtml
Counter.razor

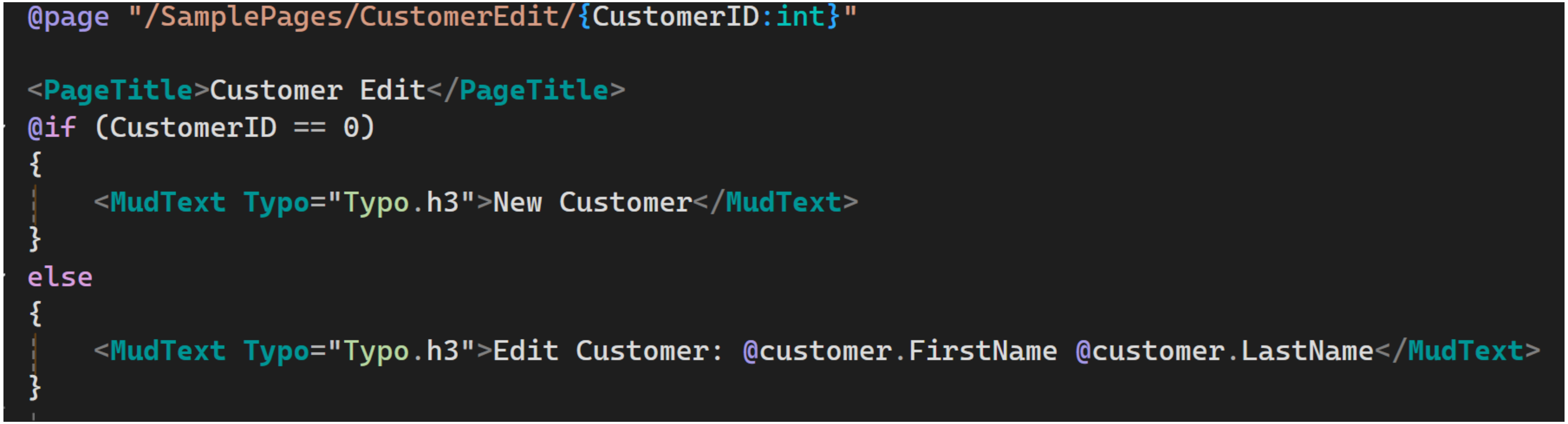
@Page Routing

- In Blazor, the **@page** directive at the top of the code defines the routing pattern for the component. In this example, the URL path **/SamplePages/CustomerEdit/{CustomerID:int}** is mapped to this component, meaning that this component will be displayed when the URL matches that pattern. The **{CustomerID:int}** portion acts as a route parameter, capturing an integer value from the URL and making it available within the component as a variable named **CustomerID**. The **:int** is a route constraint ensuring that this part of the URL must be an integer. Overall, this allows you to create a page for editing customer details, where the **CustomerID** parameter could be used to fetch or update information specific to that customer.
- For an existing customer, our routing would look like the following:
  - https://localhost:7199/SamplePages/CustomerEdit/**92**
- For a new customer, our routing would be the following:
  - https://localhost:7199/SamplePages/CustomerEdit/**0**

@Page Routing and Dynamic Heading

- If the value of “CustomerID” is zero, we create a new customer, otherwise, we edit an existing customer.

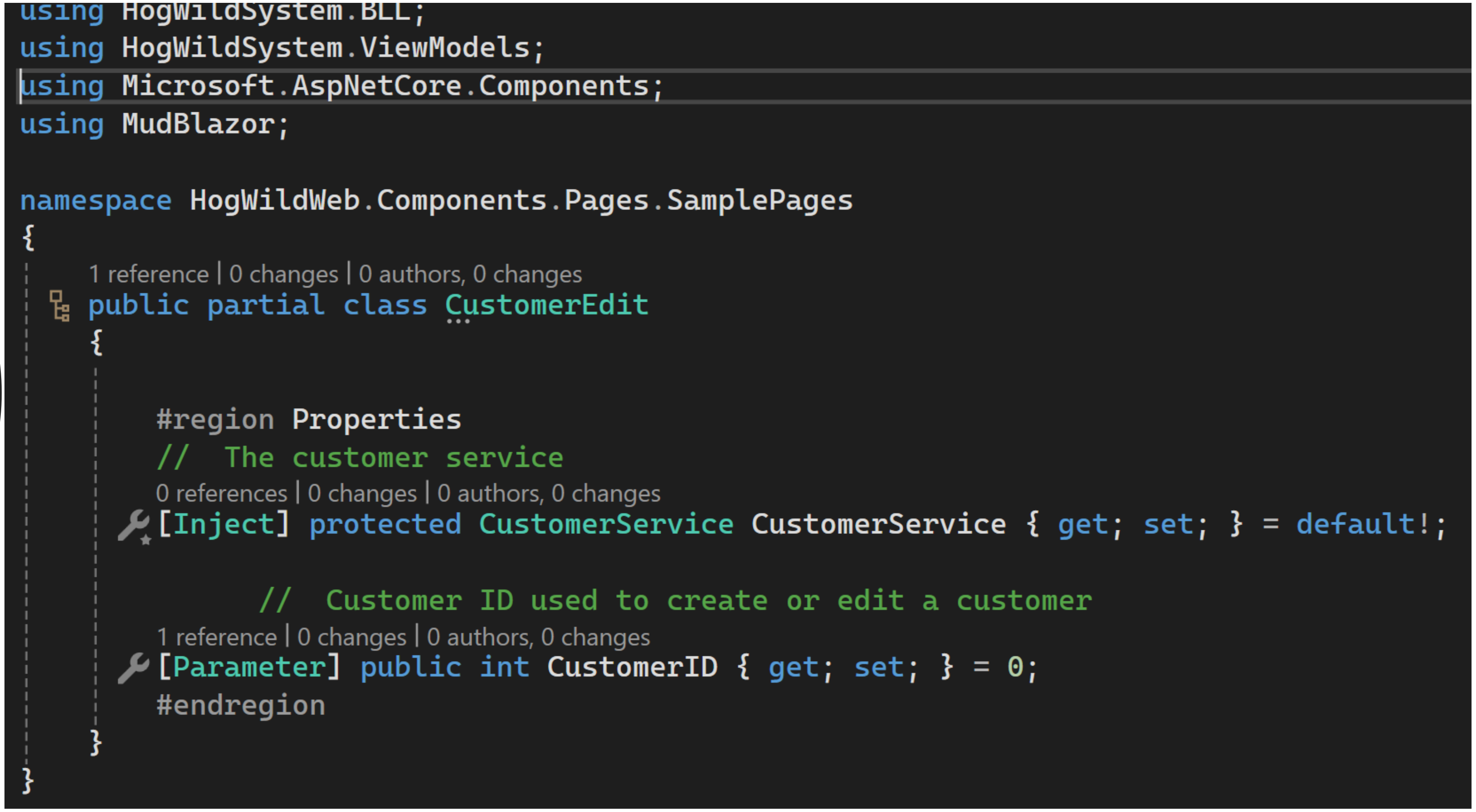
CustomerEdit.razor.cs



```
@page "/SamplePages/CustomerEdit/{CustomerID:int}"

<PageTitle>Customer Edit</PageTitle>
@if (CustomerID == 0)
{
    <MudText Typo="Typo.h3">New Customer</MudText>
}
else
{
    <MudText Typo="Typo.h3">Edit Customer: @customer.FirstName @customer.LastName</MudText>
}
```

CustomerEdit.razor.cs

Initial Code Behind

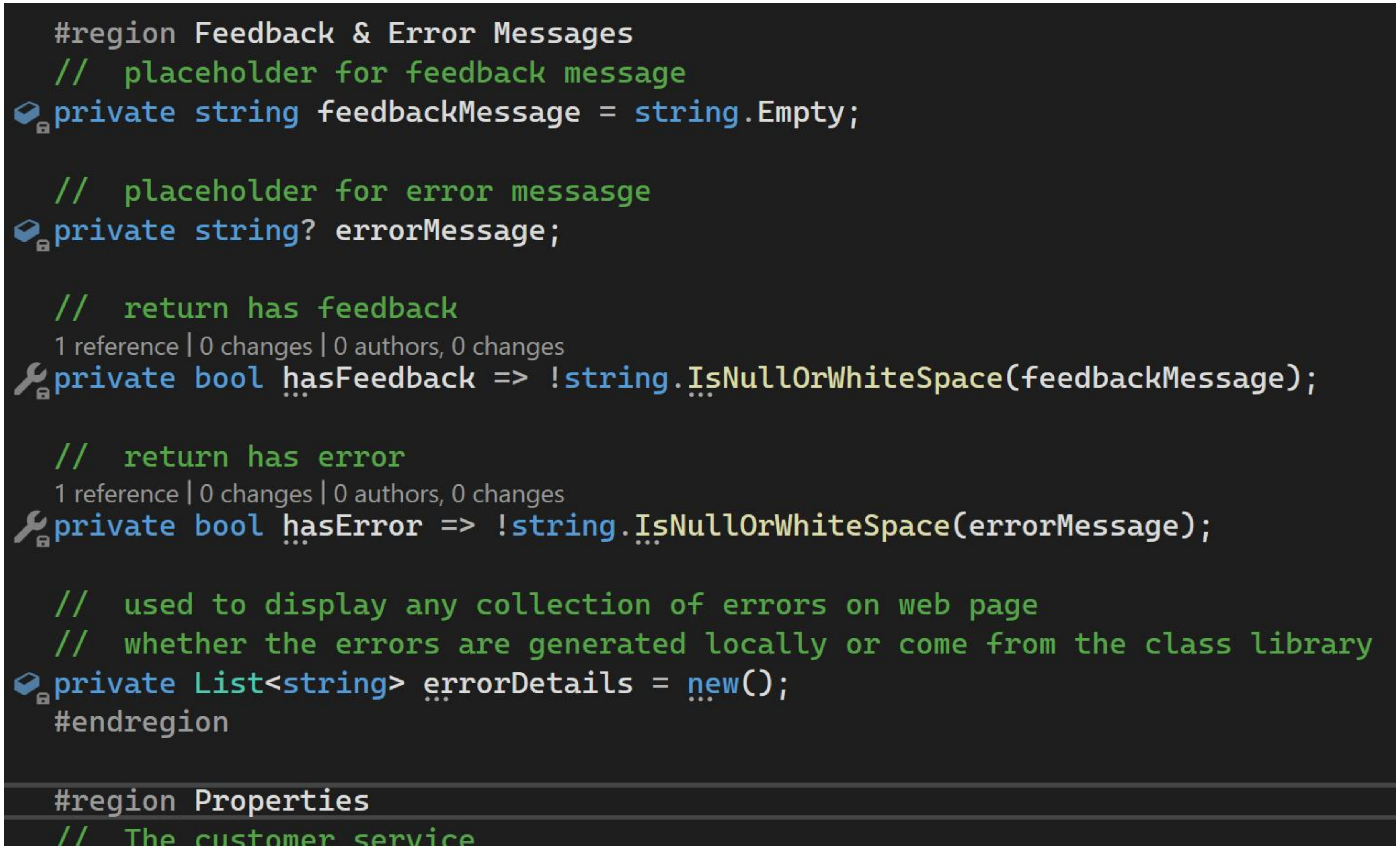


```
using HogWildSystem.BLL;
using HogWildSystem.ViewModels;
using Microsoft.AspNetCore.Components;
using MudBlazor;

namespace HogWildWeb.Components.Pages.SamplePages
{
    1 reference | 0 changes | 0 authors, 0 changes
    public partial class CustomerEdit
    {

        #region Properties
        //  The customer service
        0 references | 0 changes | 0 authors, 0 changes
        [Inject] protected CustomerService CustomerService { get; set; } = default!;

              //  Customer ID used to create or edit a customer
        1 reference | 0 changes | 0 authors, 0 changes
        [Parameter] public int CustomerID { get; set; } = 0;
        #endregion
    }
}
```

NOTE: The namespace maybe listed as either **HogWildWeb** or **HogWildWebApp**

Feedback & Error Messages

CustomerEdit.razor.cs



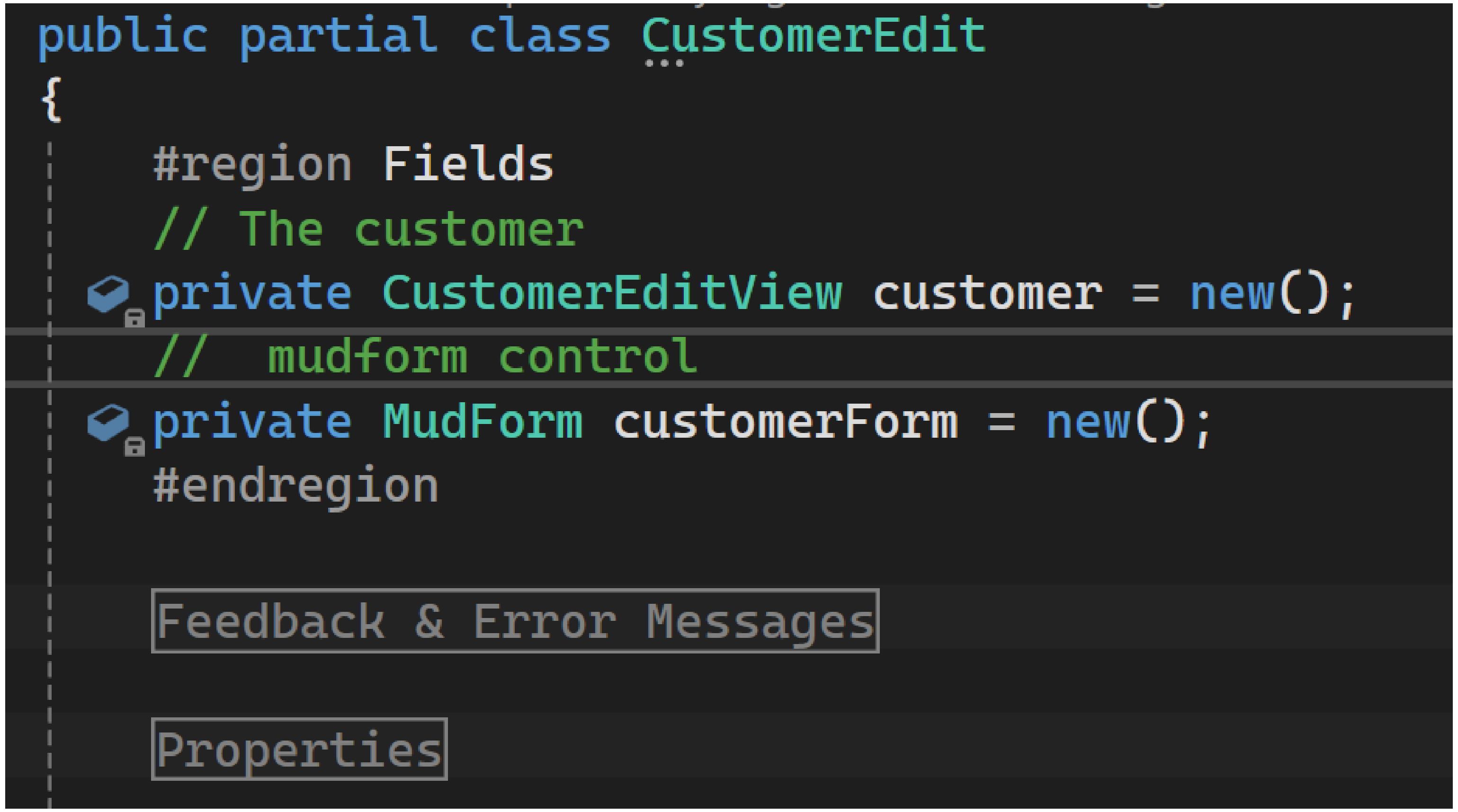
```
#region Feedback & Error Messages
//  placeholder for feedback message
private string feedbackMessage = string.Empty;

//  placeholder for error messasge
private string? errorMessage;

//  return has feedback
1 reference | 0 changes | 0 authors, 0 changes
private bool hasFeedback => !string.IsNullOrWhiteSpace(feedbackMessage);

//  return has error
1 reference | 0 changes | 0 authors, 0 changes
private bool hasError => !string.IsNullOrWhiteSpace(errorMessage);

//  used to display any collection of errors on web page
//  whether the errors are generated locally or come from the class library
private List<string> errorDetails = new();
#endregion

#region Properties
```

Coding for Displaying Minimum Customer Data (Proof of Concept)

1. Adding code behind to load a customer.
2. Add Blazor code to display first name, last name, and address #1.
3. Add code to “Customer List” to load the “Customer Editor” using the customer ID.

CustomerEdit.razor.cs

Adding Code Behind to Load a Customer

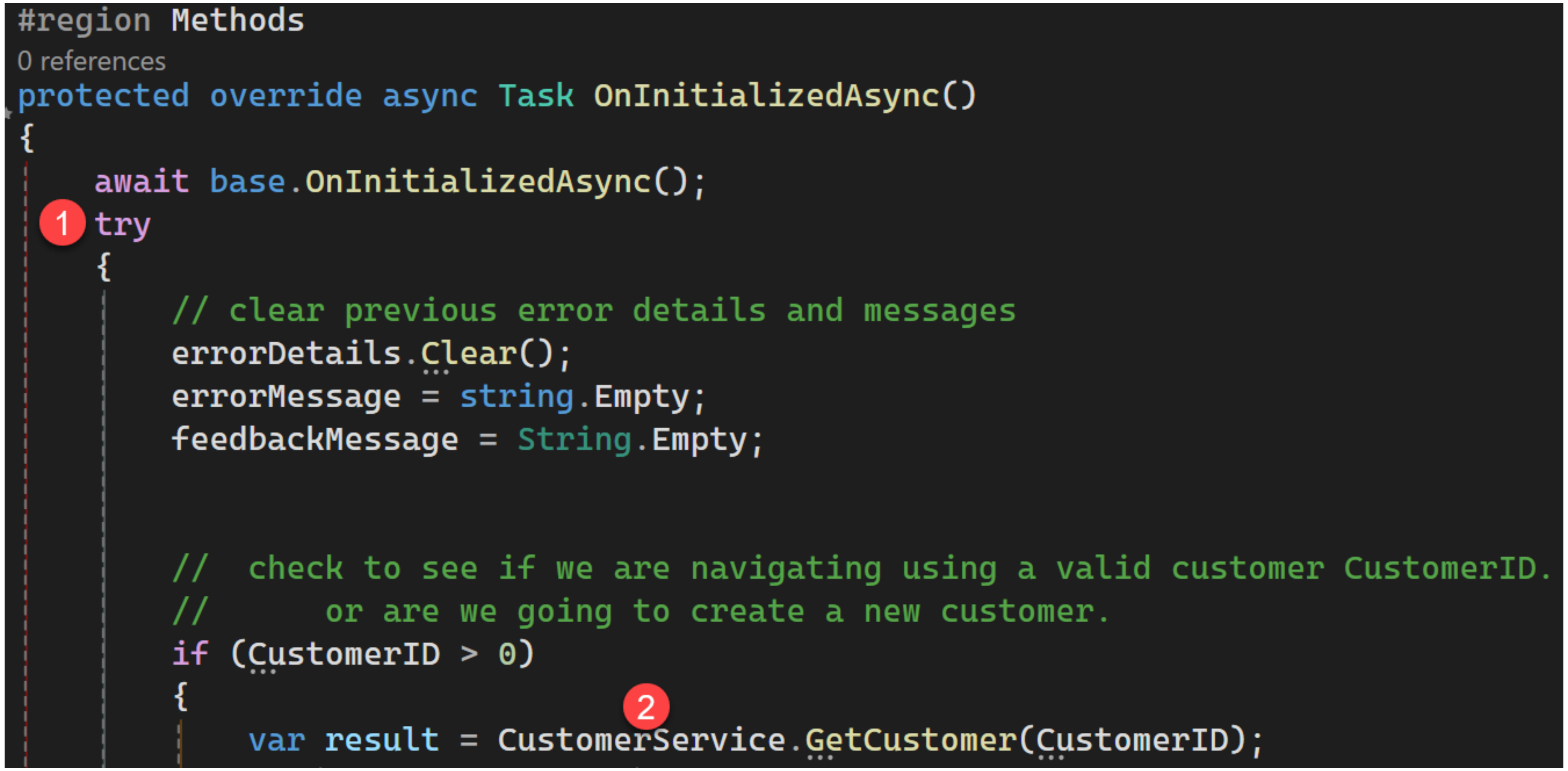


```
public partial class CustomerEdit
{
    #region Fields
    // The customer
    private CustomerEditView customer = new();
    //  mudform control
    private MudForm customerForm = new();
    #endregion

    Feedback & Error Messages

    Properties
```

CustomerEdit.razor.cs

Adding Code Behind to Load a Customer

1. Copy the code from GetCustomer.linq within your CodeBehind -> GetCustomer
2. Updated the "service class" name to "CustomerService"

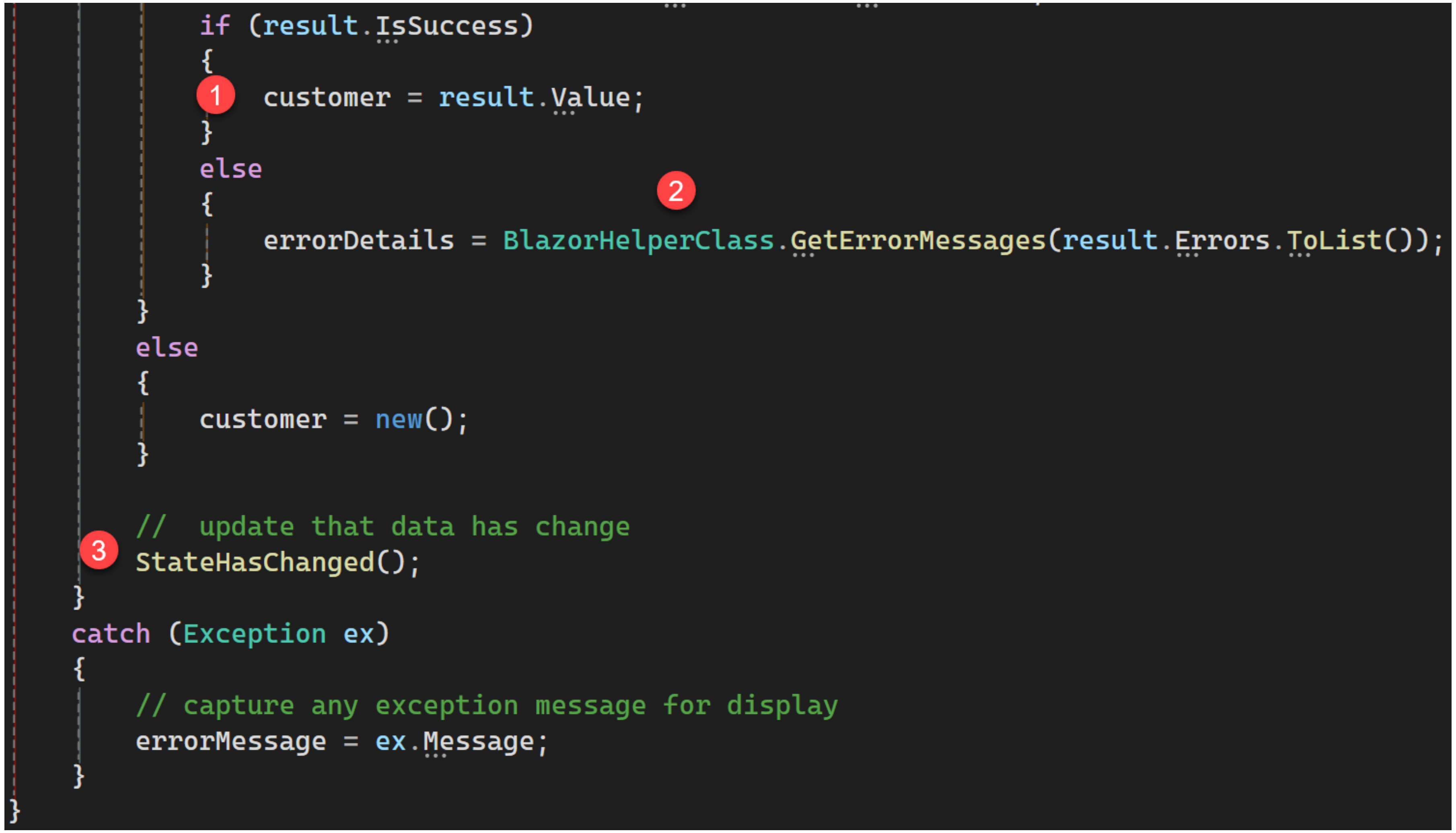


```
#region Methods
0 references
protected override async Task OnInitializedAsync()
{
    await base.OnInitializedAsync();
    try
    {
        // clear previous error details and messages
        errorDetails.Clear();
        errorMessage = string.Empty;
        feedbackMessage = String.Empty;


        //  check to see if we are navigating using a valid customer CustomerID.
        //      or are we going to create a new customer.
        if (CustomerID > 0)
        {
            var result = CustomerService.GetCustomer(CustomerID);
```

CustomerEdit.razor.cs

Adding Code Behind to Load a Customer

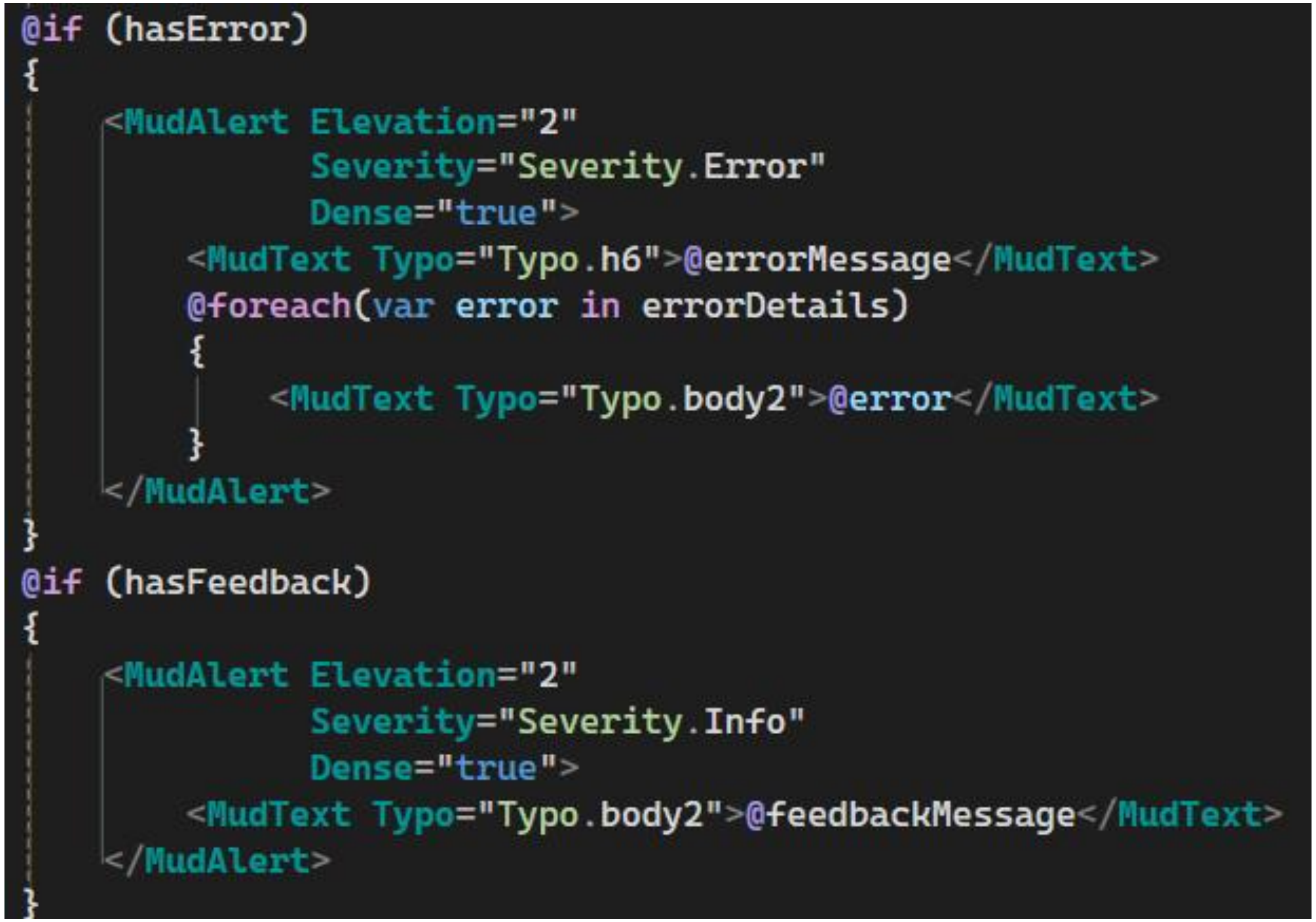
1. Rename "Customer" to "customer"
2. Add BlazorHelperClass to the GetErrorMessages
3. Add StateHasChanged



```
            if (result.IsSuccess)
            {
       1        customer = result.Value;
            }
            else
            {                                          2
                errorDetails = BlazorHelperClass.GetErrorMessages(result.Errors.ToList());
            }
        }
        else
        {
            customer = new();
        }

        //  update that data has change
   3    StateHasChanged();
    }
    catch (Exception ex)
    {
        // capture any exception message for display
        errorMessage = ex.Message;
    }
}
```

Add Error and Feedback

CustomerEdit.razor



```
@if (hasError)
{
    <MudAlert Elevation="2"
              Severity="Severity.Error"
              Dense="true">
        <MudText Typo="Typo.h6">@errorMessage</MudText>
        @foreach(var error in errorDetails)
        {
            <MudText Typo="Typo.body2">@error</MudText>
        }
    </MudAlert>
}
@if (hasFeedback)
{
    <MudAlert Elevation="2"
              Severity="Severity.Info"
              Dense="true">
        <MudText Typo="Typo.body2">@feedbackMessage</MudText>
    </MudAlert>
}
```

Validation Rules for First Name, Last Name, Address 1

- **Required**: The field cannot be left empty.

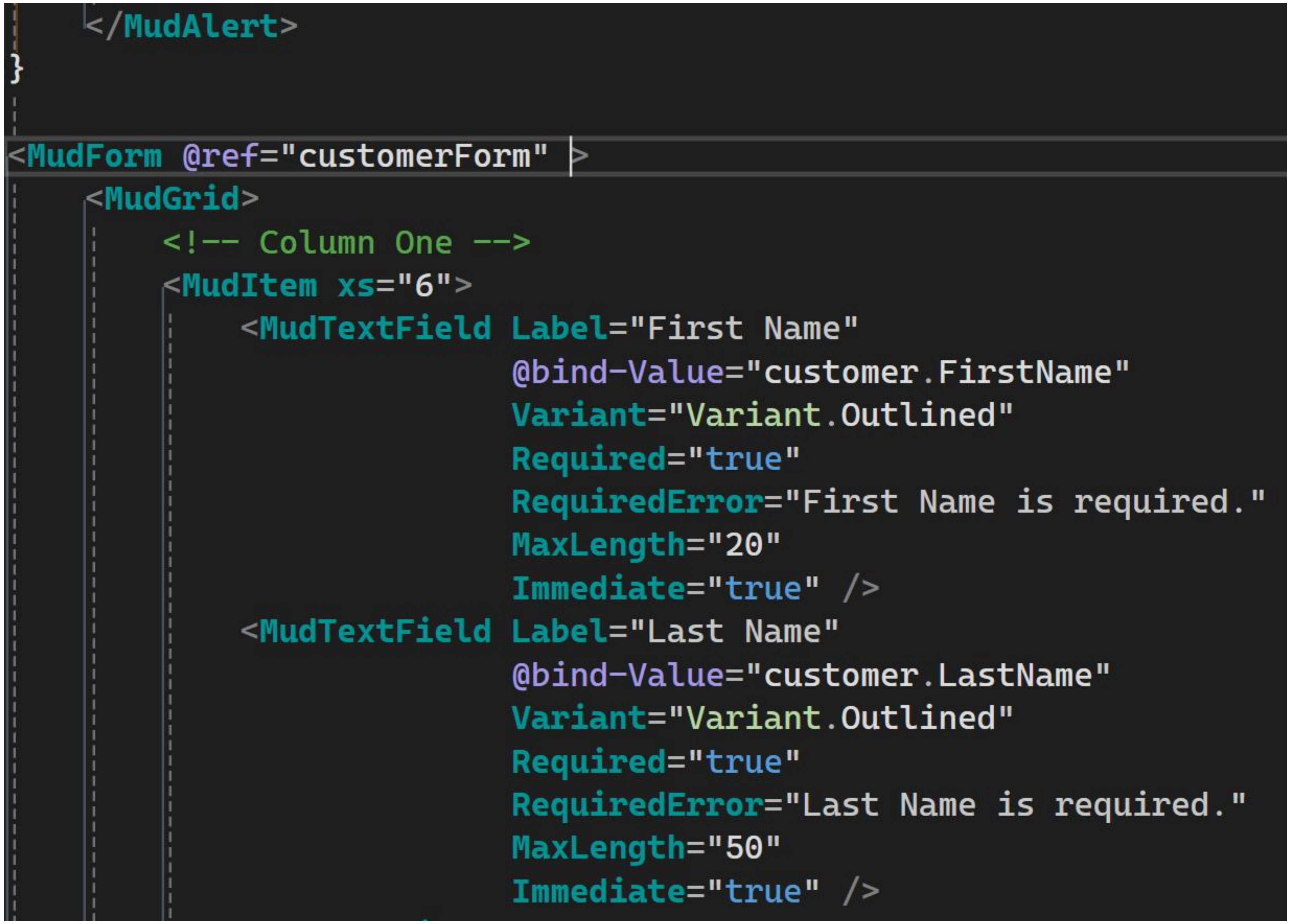
- **Error Messages**:
  - *First Name is required.*
  - *Last Name is required.*
  - *Address 1 is required.*

- **Maximum Length**:
  - *First Name: 20 characters*
  - *Last Name: 50 characters*
  - *Address 1: 50 characters*

- **Immediate Validation**: Errors appear as soon as the user enters or removes text.

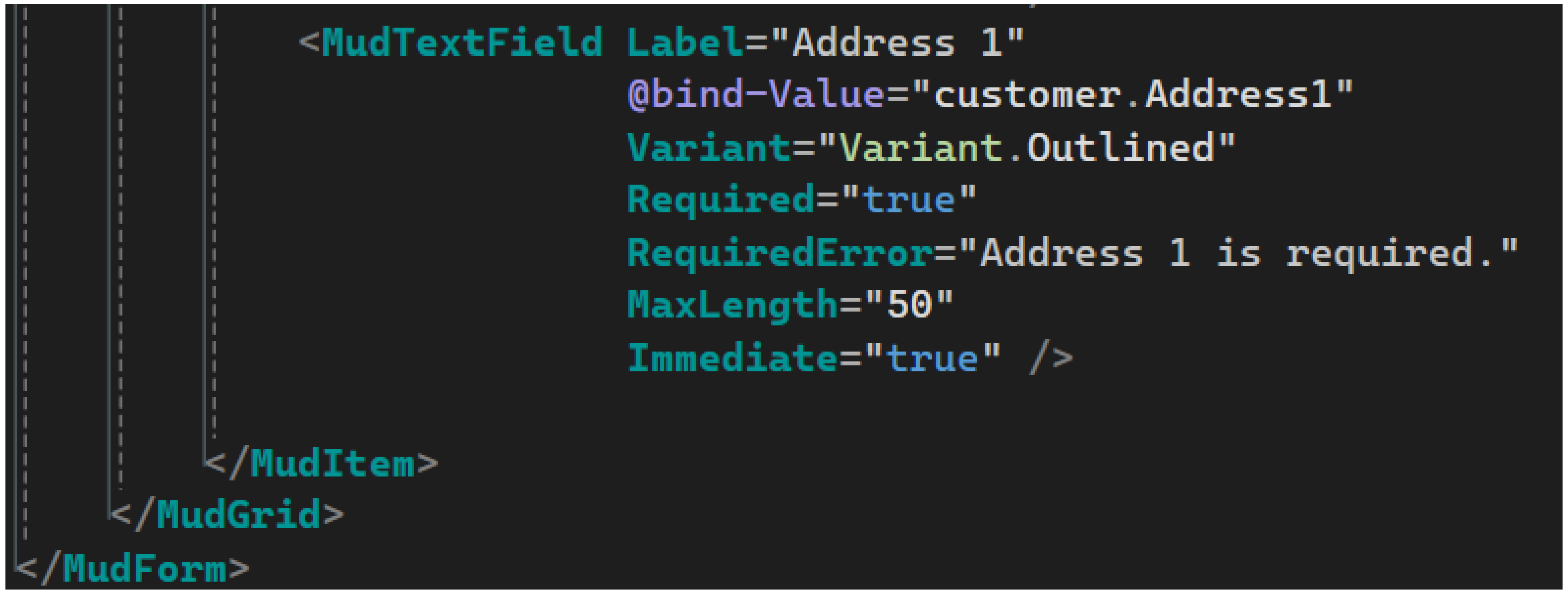
Add Blazor Code to Display First Three Fields

CustomerEdit.razor



```
    </MudAlert>
}

<MudForm @ref="customerForm" >
    <MudGrid>
        <!-- Column One -->
        <MudItem xs="6">
            <MudTextField Label="First Name"
                          @bind-Value="customer.FirstName"
                          Variant="Variant.Outlined"
                          Required="true"
                          RequiredError="First Name is required."
                          MaxLength="20"
                          Immediate="true" />
            <MudTextField Label="Last Name"
                          @bind-Value="customer.LastName"
                          Variant="Variant.Outlined"
                          Required="true"
                          RequiredError="Last Name is required."
                          MaxLength="50"
                          Immediate="true" />
```

Add Blazor Code to Display First Three Fields

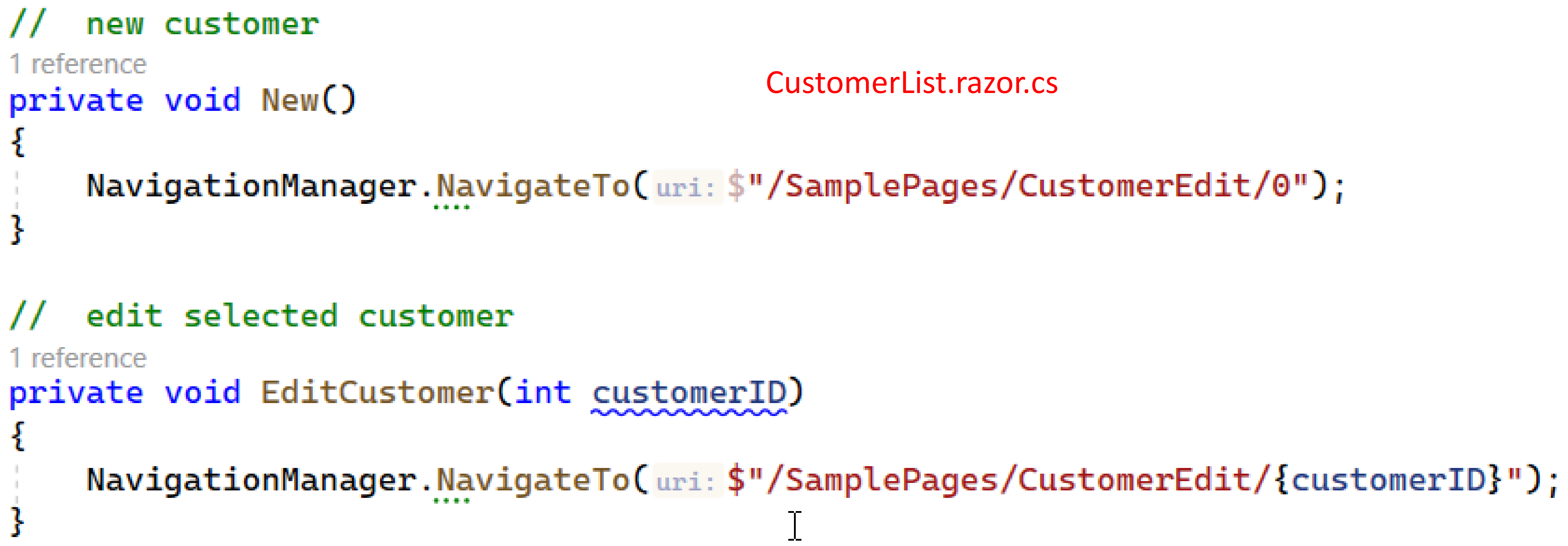


```
            <MudTextField Label="Address 1"
                          @bind-Value="customer.Address1"
                          Variant="Variant.Outlined"
                          Required="true"
                          RequiredError="Address 1 is required."
                          MaxLength="50"
                          Immediate="true" />

        </MudItem>
    </MudGrid>
</MudForm>
```

CustomerEdit.razor

Navigation to Customer Editor From Customer List

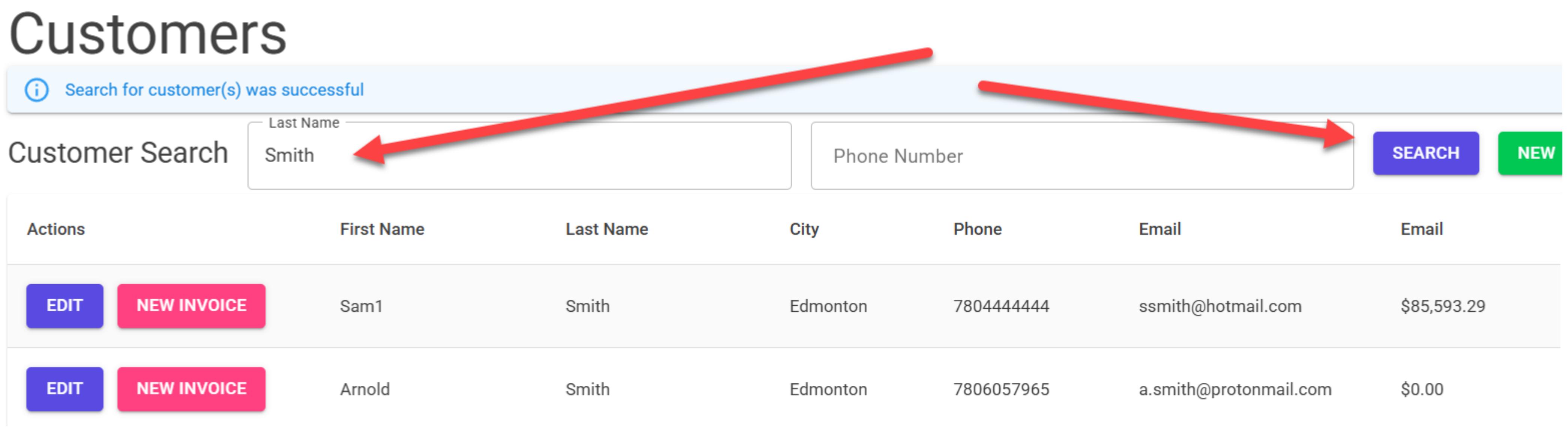
- NOTE: The URL should be the location of your "Customer Edit" page
  - /SamplePages/CustomerEdit/…



CustomerList.razor.cs

```
//  new customer
1 reference
private void New()
{
    NavigationManager.NavigateTo(uri: $"/SamplePages/CustomerEdit/0");
}

//  edit selected customer
1 reference
private void EditCustomer(int customerID)
{
    NavigationManager.NavigateTo(uri: $"/SamplePages/CustomerEdit/{customerID}");
}
```

Blazor Page Output (MudGrid)



Customers

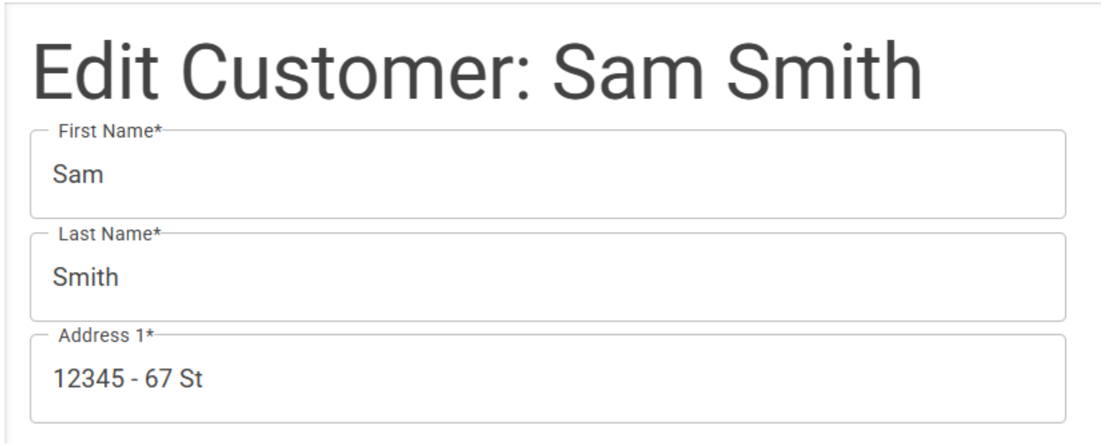
Search for customer(s) was successful

Customer Search

Last Name: Smith

Phone Number

Actions	First Name	Last Name	City	Phone	Email	Email
EDIT NEW INVOICE	Sam1	Smith	Edmonton	7804444444	ssmith@hotmail.com	$85,593.29
EDIT NEW INVOICE	Arnold	Smith	Edmonton	7806057965	a.smith@protonmail.com	$0.00



Edit Customer: Sam Smith

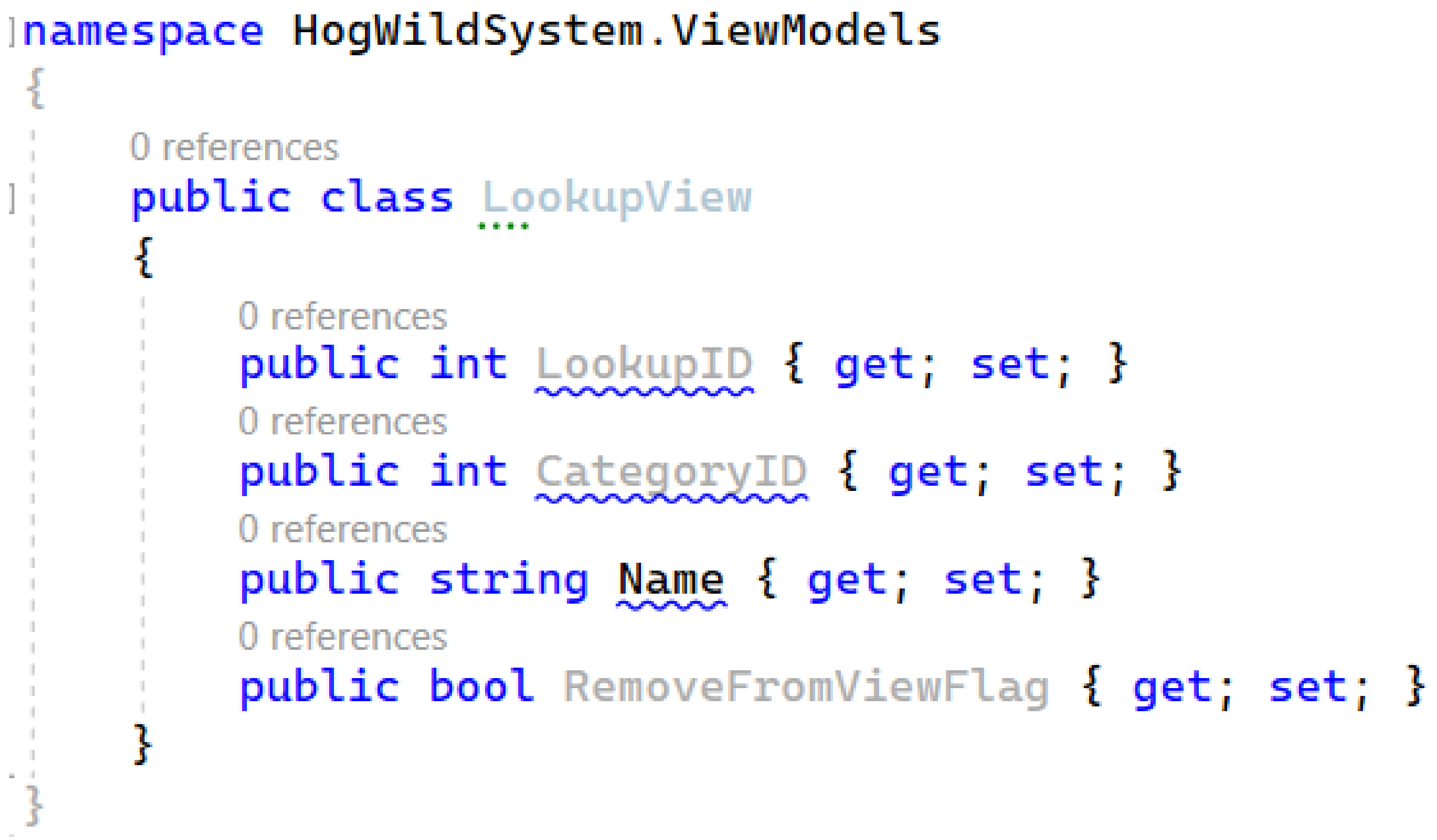
First Name*	Sam
Last Name*	Smith
Address 1*	12345 - 67 St

Lookups

- The “Customer Editor” has three dropdown lists that are used to populate the customer data
  - Province
  - Country
  - Customer Status
- The following items need to be created for this to happen.
  - A “Lookup” view model.
  - A “Category/Lookup Service” that will accept a category and return a list of values associated with the category.
  - Adding “Category/Lookup Service” to “HogWildExtension.cs” as a Transient.
  - Populate the Lookup lists when the Blazor page is created.
  - Add the “Lookup” list to the Blazor page.

Lookup View Model

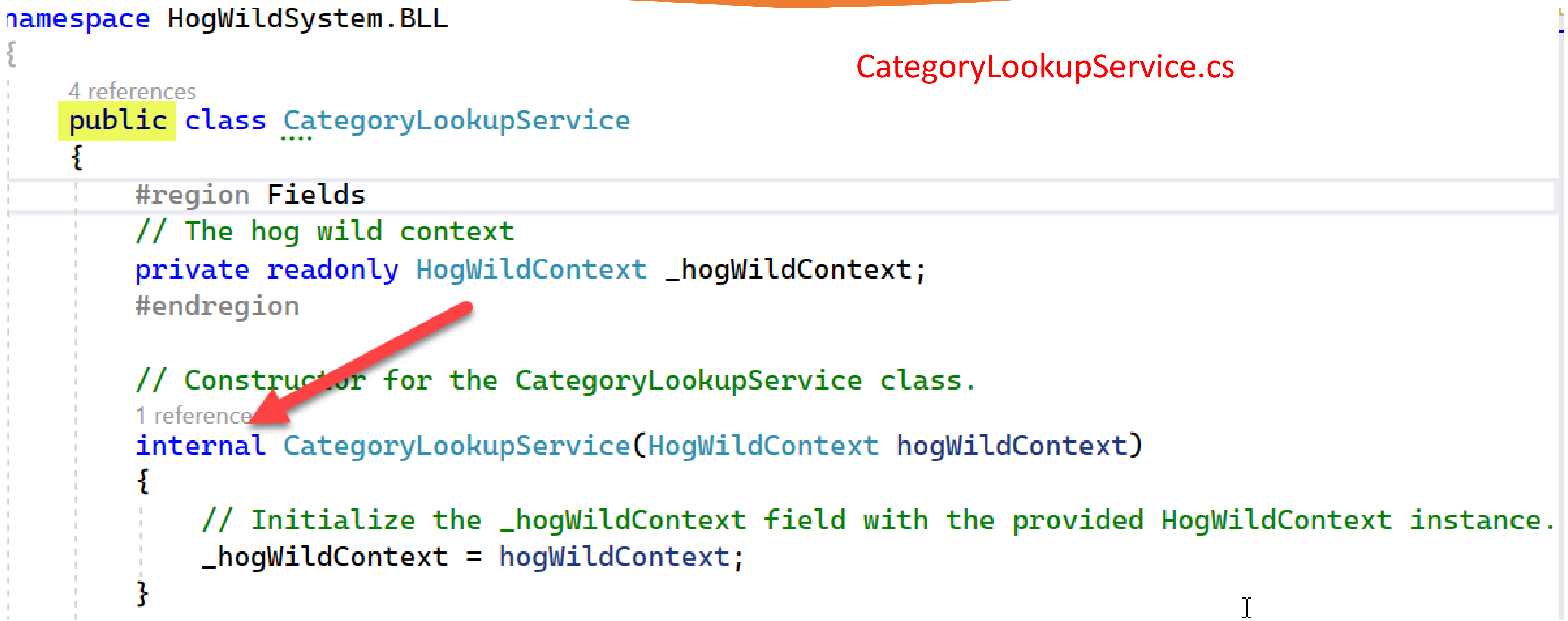
LookupView.cs



```
namespace HogWildSystem.ViewModels
{
    0 references
    public class LookupView
    {
        0 references
        public int LookupID { get; set; }
        0 references
        public int CategoryID { get; set; }
        0 references
        public string Name { get; set; }
        0 references
        public bool RemoveFromViewFlag { get; set; }
    }
}
```

Category/Lookup Service

- NOTE: Normally, we would also include the CRUD function for “Category”, but we will only focus on the “Lookup” at this time.

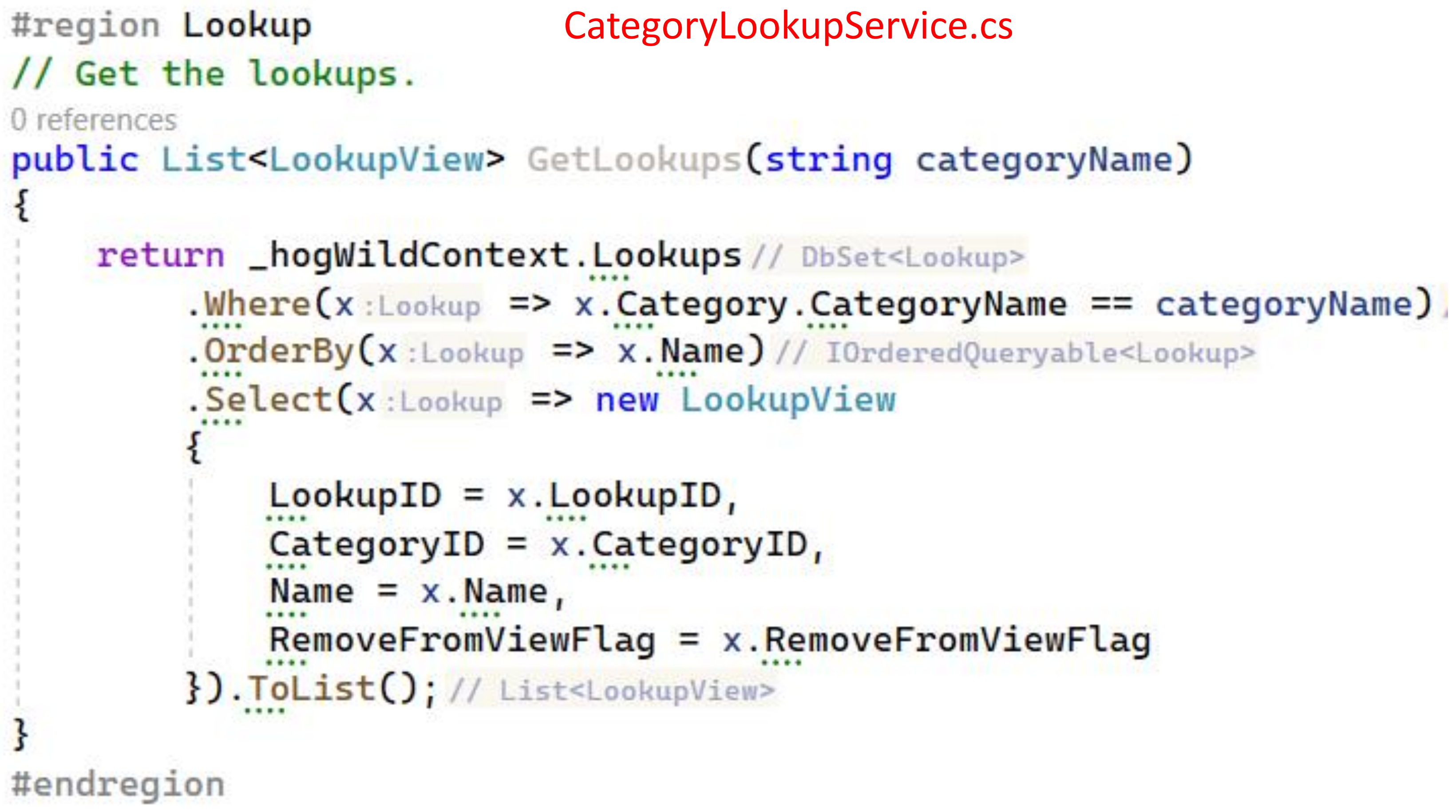


CategoryLookupService.cs

```
namespace HogWildSystem.BLL
{
    4 references
    public class CategoryLookupService
    {
        #region Fields
        // The hog wild context
        private readonly HogWildContext _hogWildContext;
        #endregion

        // Constructor for the CategoryLookupService class.
        1 reference
        internal CategoryLookupService(HogWildContext hogWildContext)
        {
            // Initialize the _hogWildContext field with the provided HogWildContext instance.
            _hogWildContext = hogWildContext;
        }
```

Category/Lookup Service (Continue)

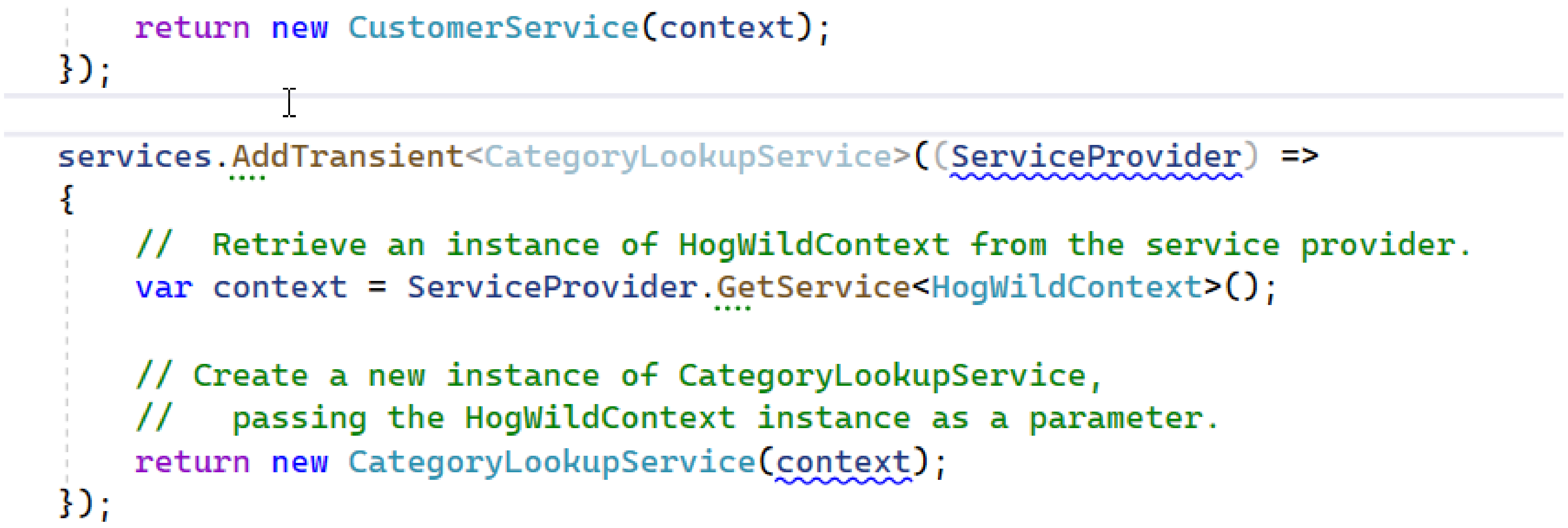
CategoryLookupService.cs



```
#region Lookup
// Get the lookups.
0 references
public List<LookupView> GetLookups(string categoryName)
{
    return _hogWildContext.Lookups // DbSet<Lookup>
        .Where(x :Lookup  => x.Category.CategoryName == categoryName)
        .OrderBy(x :Lookup  => x.Name) // IOrderedQueryable<Lookup>
        .Select(x :Lookup  => new LookupView
        {
            LookupID = x.LookupID,
            CategoryID = x.CategoryID,
            Name = x.Name,
            RemoveFromViewFlag = x.RemoveFromViewFlag
        }).ToList(); // List<LookupView>
}
#endregion
```

Add Transient Category/Lookup Service

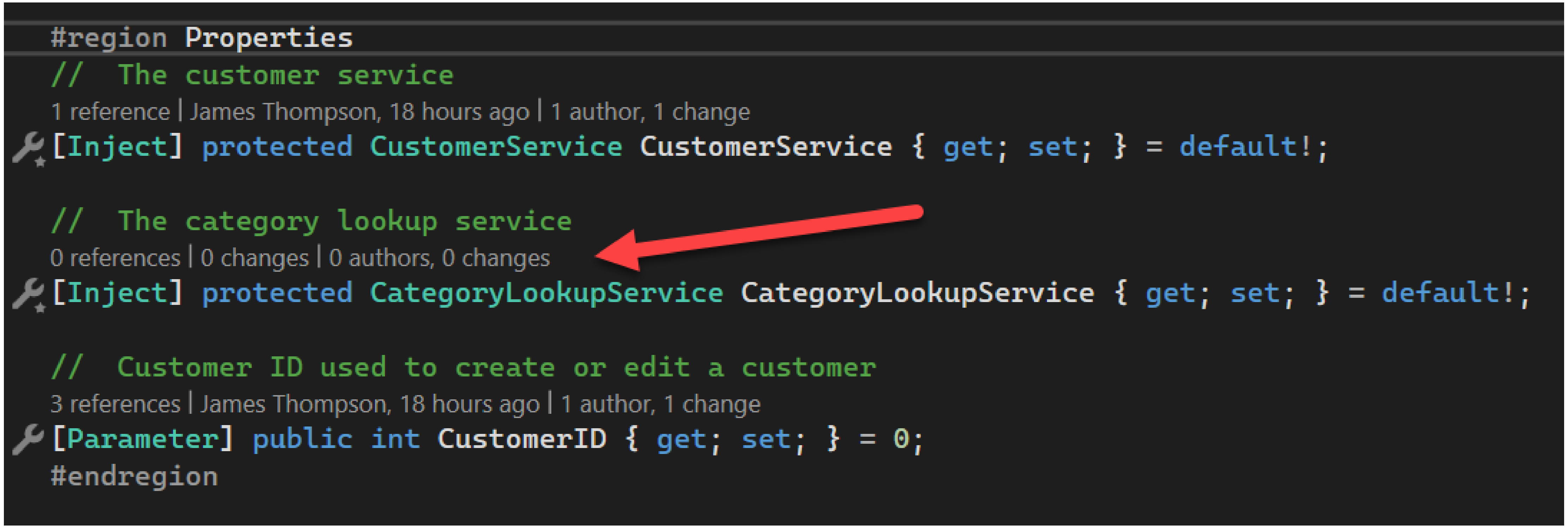
HogWildExtension.cs



```
    return new CustomerService(context);
});

services.AddTransient<CategoryLookupService>((ServiceProvider) =>
{
    //  Retrieve an instance of HogWildContext from the service provider.
    var context = ServiceProvider.GetService<HogWildContext>();

    // Create a new instance of CategoryLookupService,
    //   passing the HogWildContext instance as a parameter.
    return new CategoryLookupService(context);
});
```

Add Category Service

CustomerEdit.razor.cs

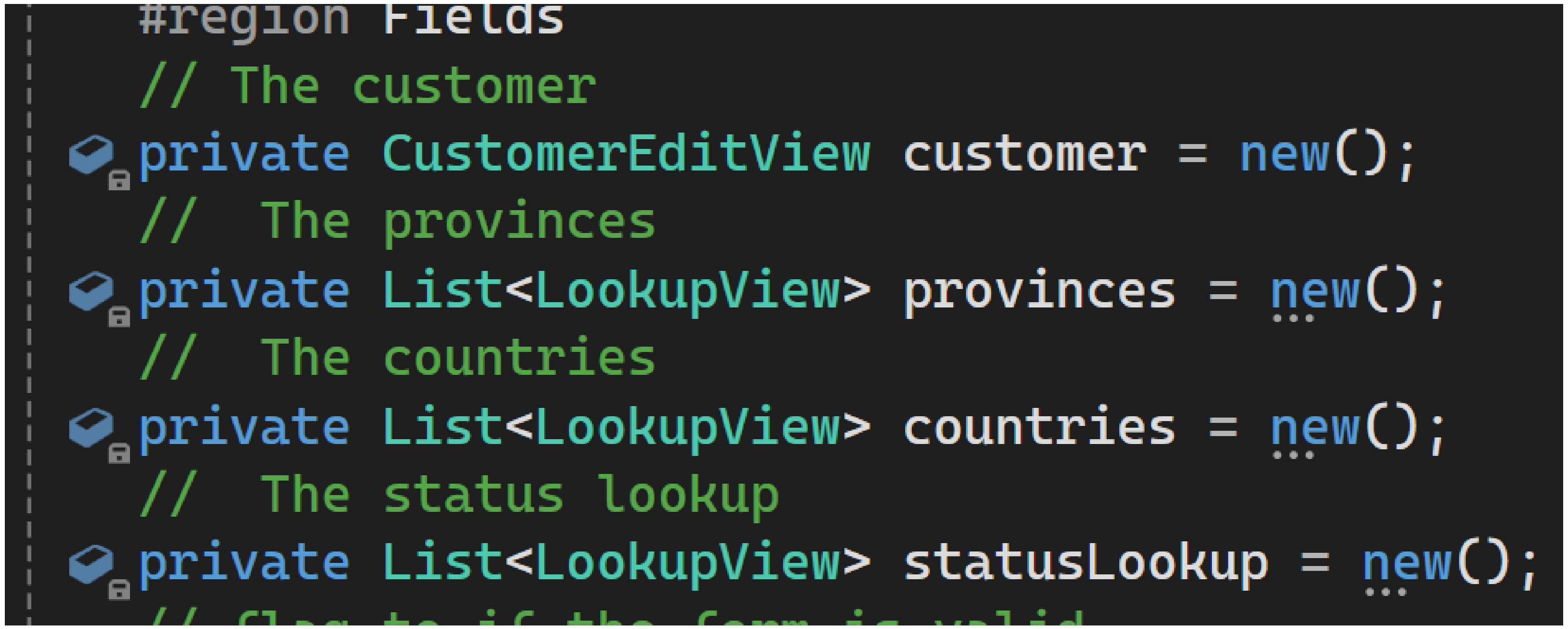


```
#region Properties
//  The customer service
1 reference | James Thompson, 18 hours ago | 1 author, 1 change
[Inject] protected CustomerService CustomerService { get; set; } = default!;

//  The category lookup service
0 references | 0 changes | 0 authors, 0 changes
[Inject] protected CategoryLookupService CategoryLookupService { get; set; } = default!;

//  Customer ID used to create or edit a customer
3 references | James Thompson, 18 hours ago | 1 author, 1 change
[Parameter] public int CustomerID { get; set; } = 0;
#endregion
```

Populate Lookup Lists

- Add List<LookupView> for each list.

CustomerEdit.razor.cs

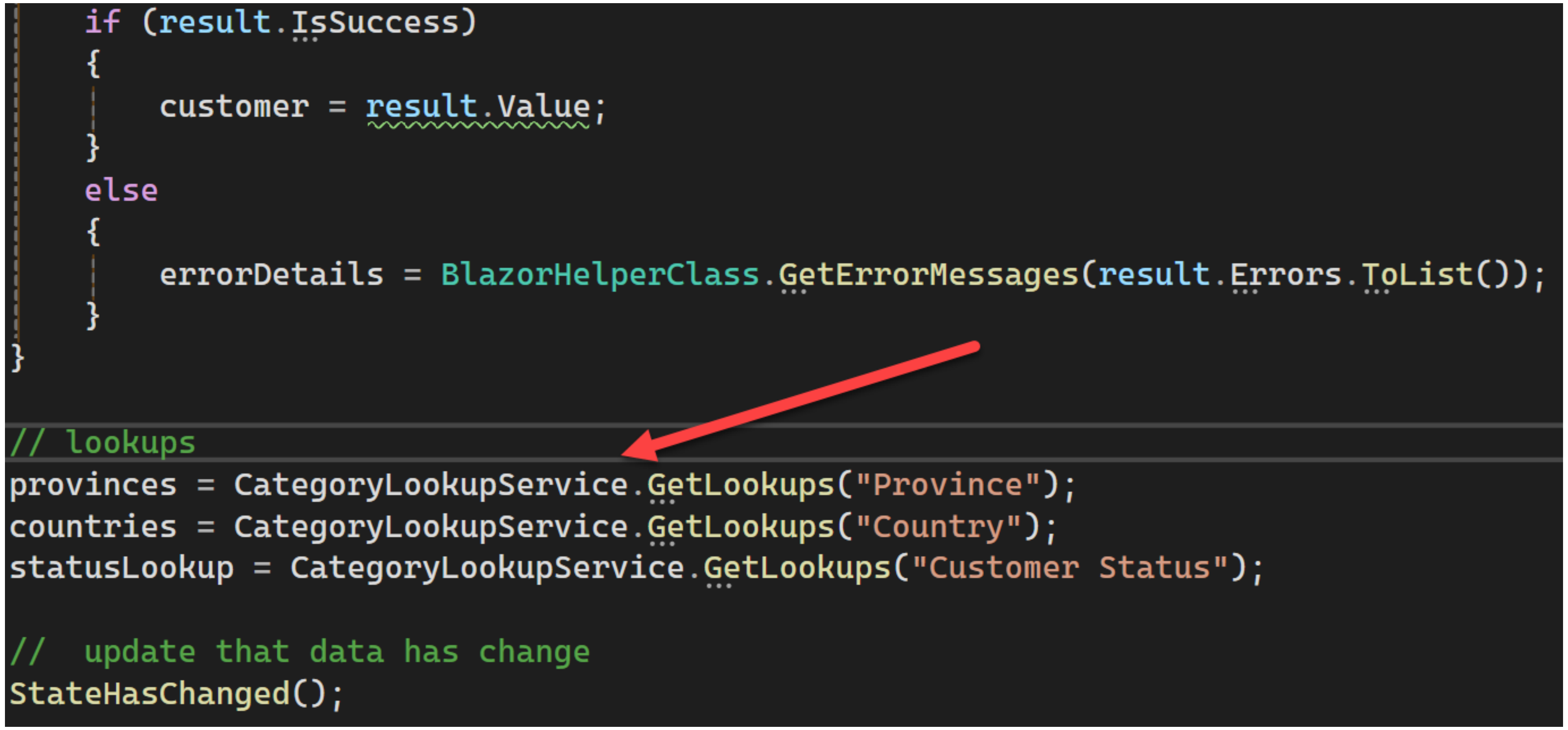


```
#region Fields
// The customer
private CustomerEditView customer = new();
//  The provinces
private List<LookupView> provinces = new();
//  The countries
private List<LookupView> countries = new();
//  The status lookup
private List<LookupView> statusLookup = new();
// flag to if the form is valid
```

Populate Lookup Lists

Add the populating of the list to the OnInitializedAsync method.

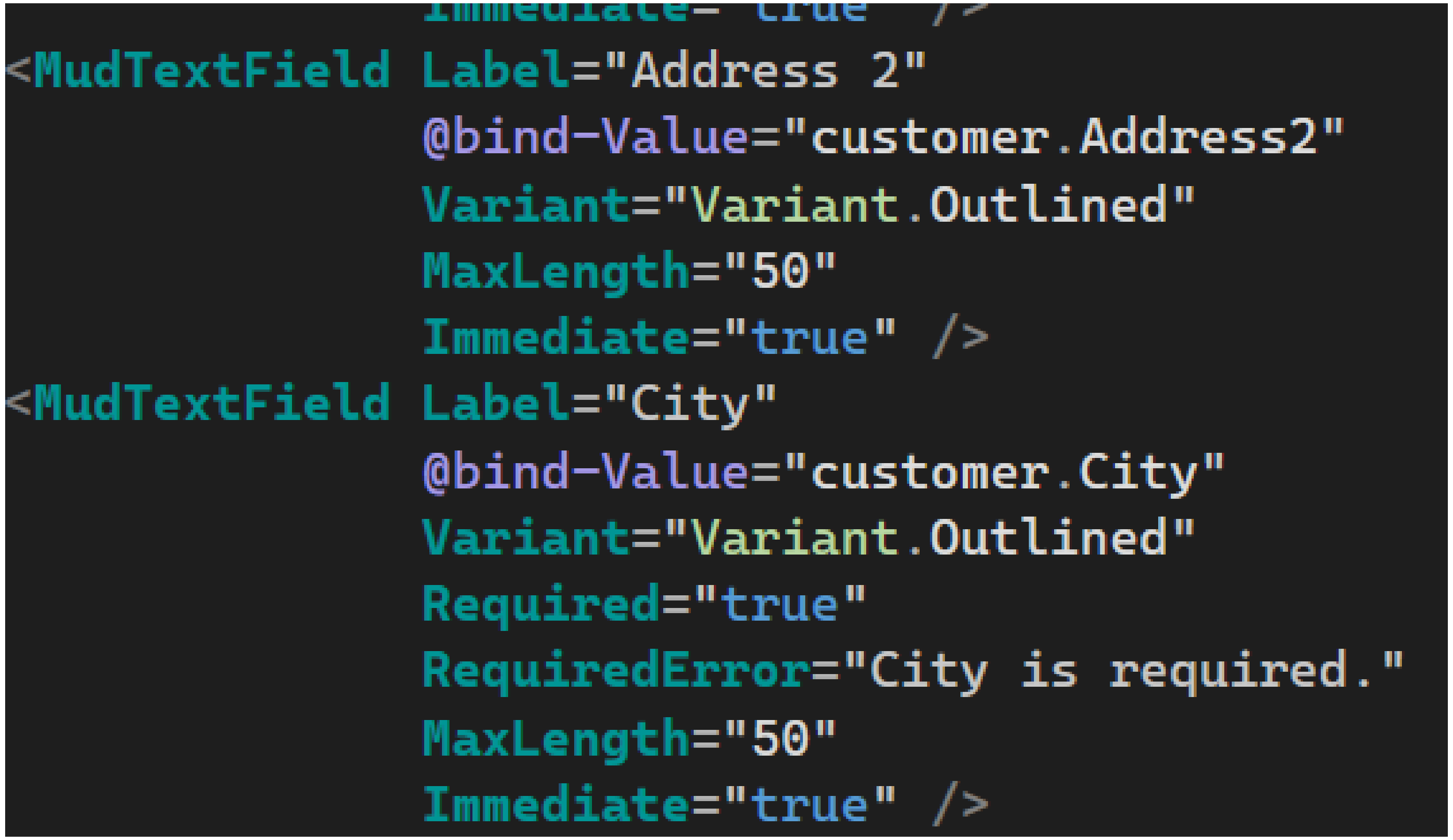
CustomerEdit.razor.cs



```
    if (result.IsSuccess)
    {
        customer = result.Value;
    }
    else
    {
        errorDetails = BlazorHelperClass.GetErrorMessages(result.Errors.ToList());
    }
}

// lookups
provinces = CategoryLookupService.GetLookups("Province");
countries = CategoryLookupService.GetLookups("Country");
statusLookup = CategoryLookupService.GetLookups("Customer Status");

//  update that data has change
StateHasChanged();
```

Updating Blazor Page

CustomerEdit.razor

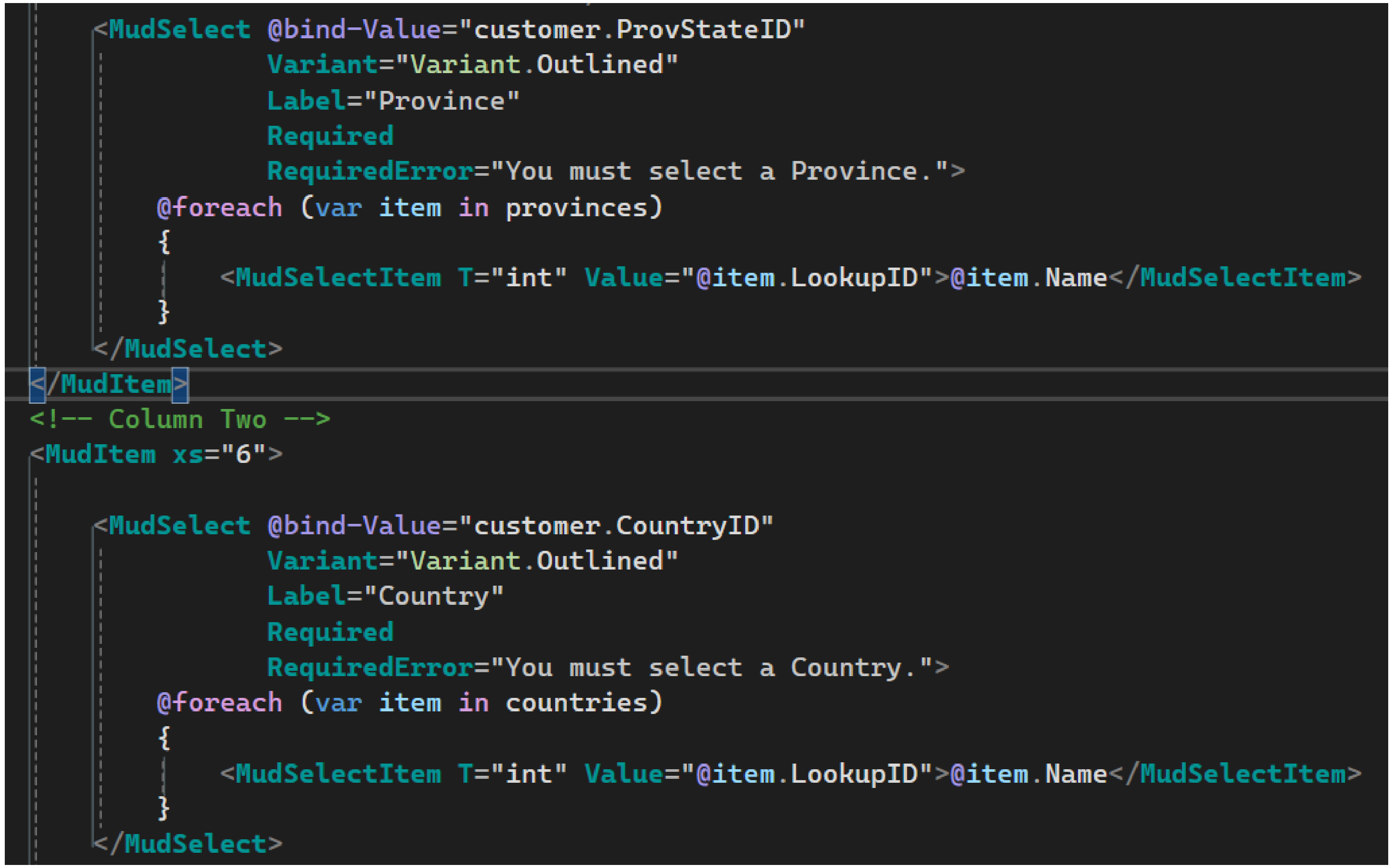


```
              Immediate="true" />
<MudTextField Label="Address 2"
              @bind-Value="customer.Address2"
              Variant="Variant.Outlined"
              MaxLength="50"
              Immediate="true" />
<MudTextField Label="City"
              @bind-Value="customer.City"
              Variant="Variant.Outlined"
              Required="true"
              RequiredError="City is required."
              MaxLength="50"
              Immediate="true" />
```

Updating Blazor Page (Continue)

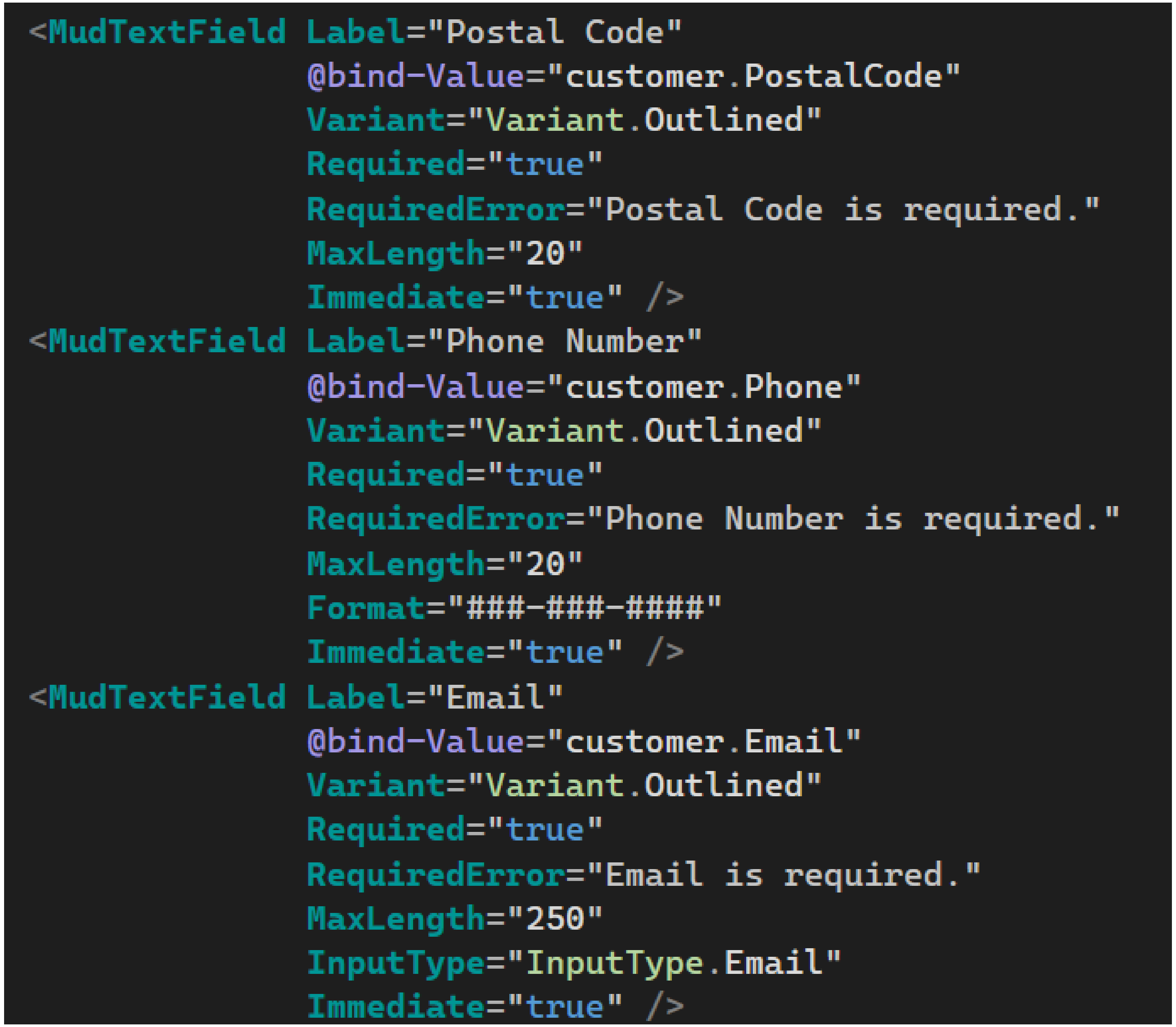
CustomerEdit.razor

- Create a new column for customer data.
- Add Dropdown list for Province & Country



```
        <MudSelect @bind-Value="customer.ProvStateID"
                   Variant="Variant.Outlined"
                   Label="Province"
                   Required
                   RequiredError="You must select a Province.">
            @foreach (var item in provinces)
            {
                <MudSelectItem T="int" Value="@item.LookupID">@item.Name</MudSelectItem>
            }
        </MudSelect>
    </MudItem>
    <!-- Column Two -->
    <MudItem xs="6">

        <MudSelect @bind-Value="customer.CountryID"
                   Variant="Variant.Outlined"
                   Label="Country"
                   Required
                   RequiredError="You must select a Country.">
            @foreach (var item in countries)
            {
                <MudSelectItem T="int" Value="@item.LookupID">@item.Name</MudSelectItem>
            }
        </MudSelect>
```

Updating Blazor Page (Continue)

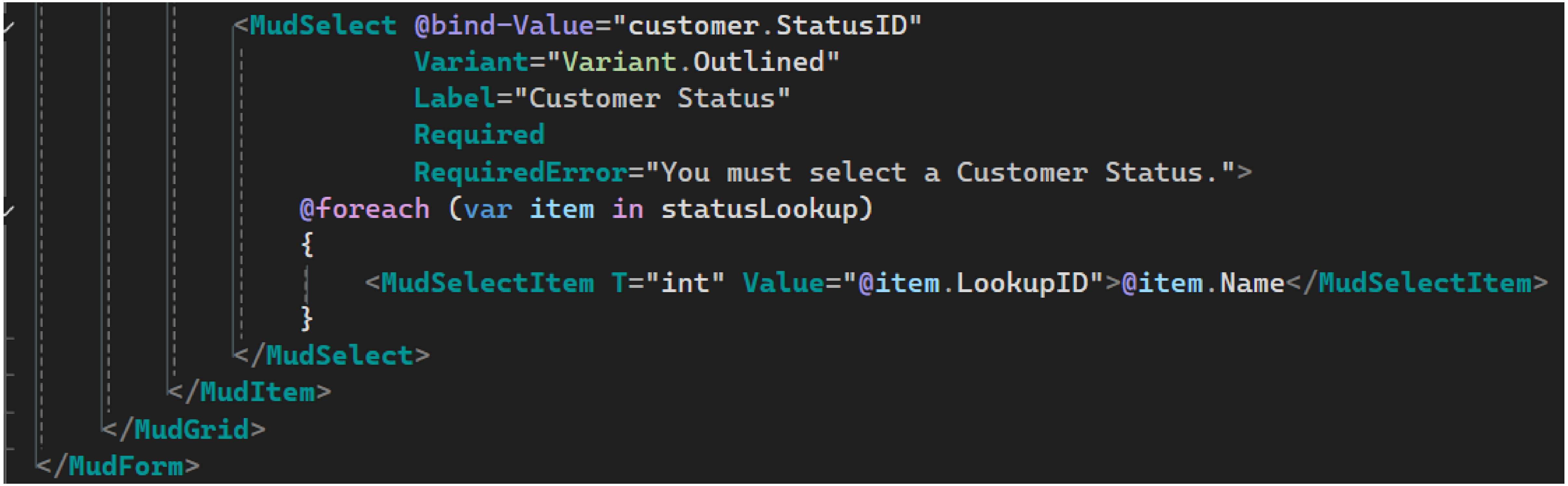


```
<MudTextField Label="Postal Code"
              @bind-Value="customer.PostalCode"
              Variant="Variant.Outlined"
              Required="true"
              RequiredError="Postal Code is required."
              MaxLength="20"
              Immediate="true" />
<MudTextField Label="Phone Number"
              @bind-Value="customer.Phone"
              Variant="Variant.Outlined"
              Required="true"
              RequiredError="Phone Number is required."
              MaxLength="20"
              Format="###-###-####"
              Immediate="true" />
<MudTextField Label="Email"
              @bind-Value="customer.Email"
              Variant="Variant.Outlined"
              Required="true"
              RequiredError="Email is required."
              MaxLength="250"
              InputType="InputType.Email"
              Immediate="true" />
```

CustomerEdit.razor

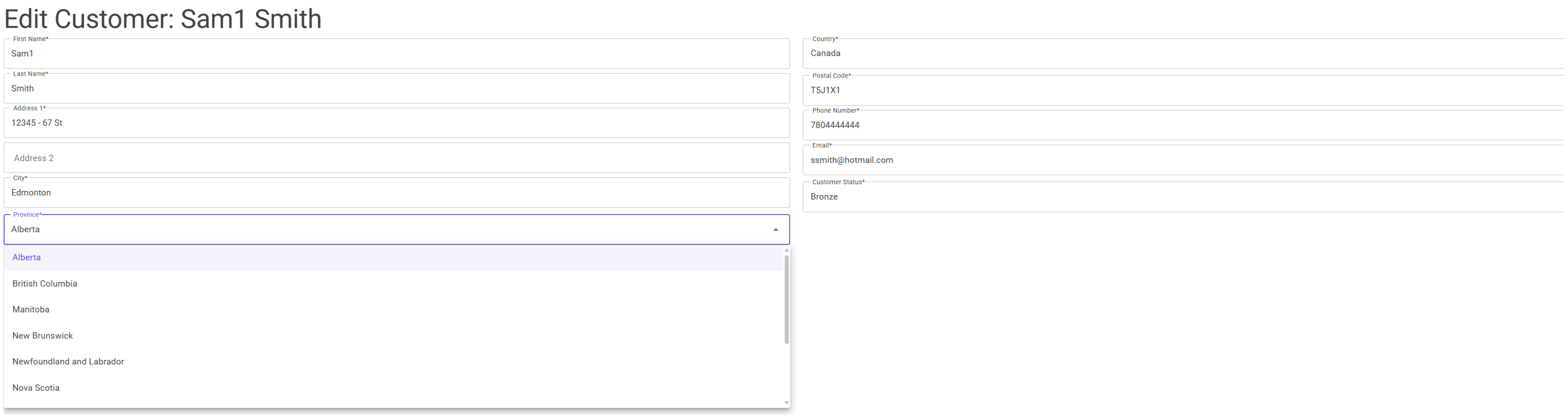
Updating Blazor Page (Continue)

CustomerEdit.razor



```
                <MudSelect @bind-Value="customer.StatusID"
                           Variant="Variant.Outlined"
                           Label="Customer Status"
                           Required
                           RequiredError="You must select a Customer Status.">
                    @foreach (var item in statusLookup)
                    {
                        <MudSelectItem T="int" Value="@item.LookupID">@item.Name</MudSelectItem>
                    }
                </MudSelect>
            </MudItem>
        </MudGrid>
    </MudForm>
```

Blazor Page Output (MudGrid)



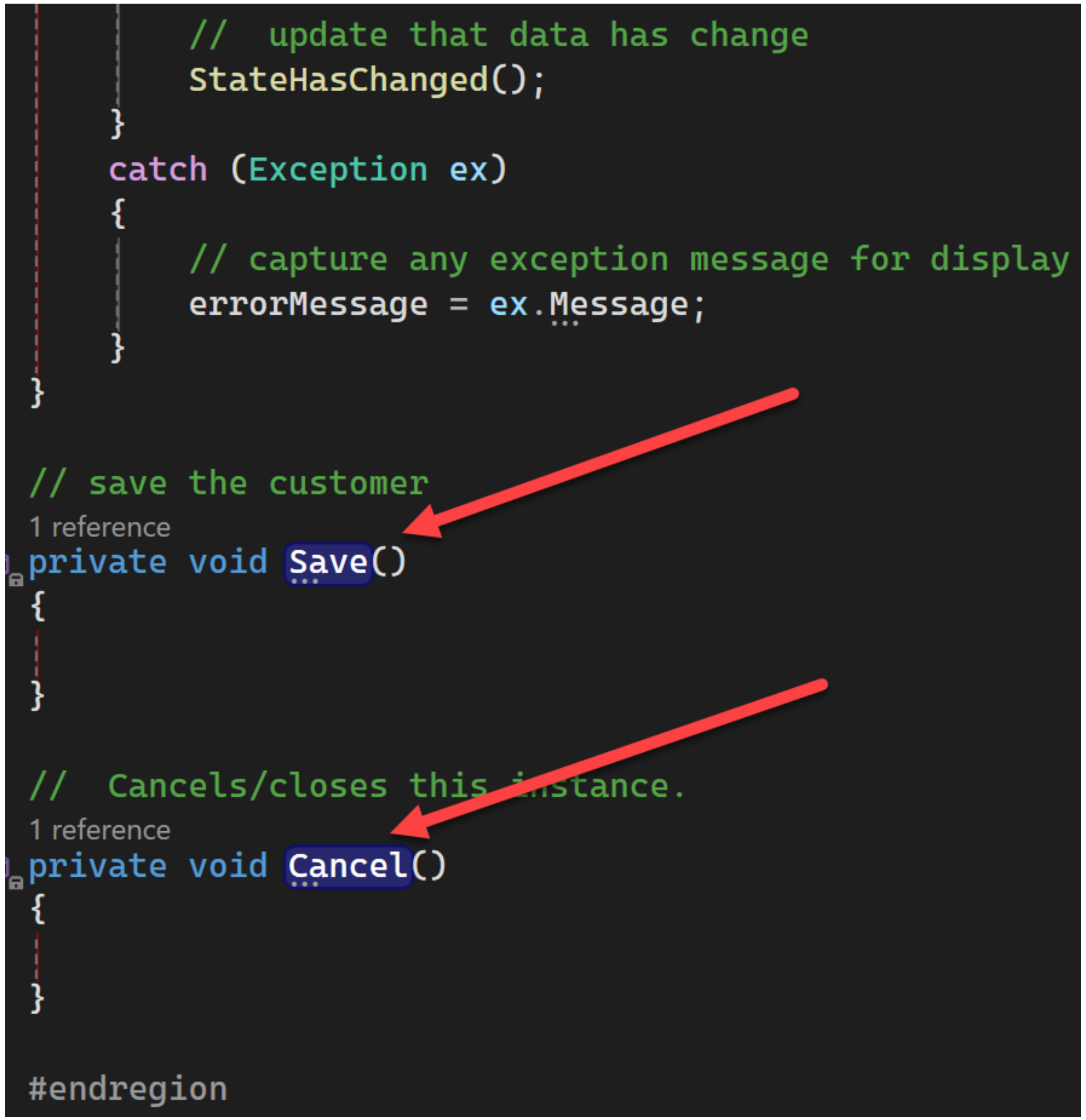
Edit Customer: Sam1 Smith

First Name*: Sam1
Last Name*: Smith
Address 1*: 12345 - 67 St
Address 2
City*: Edmonton
Province*: Alberta

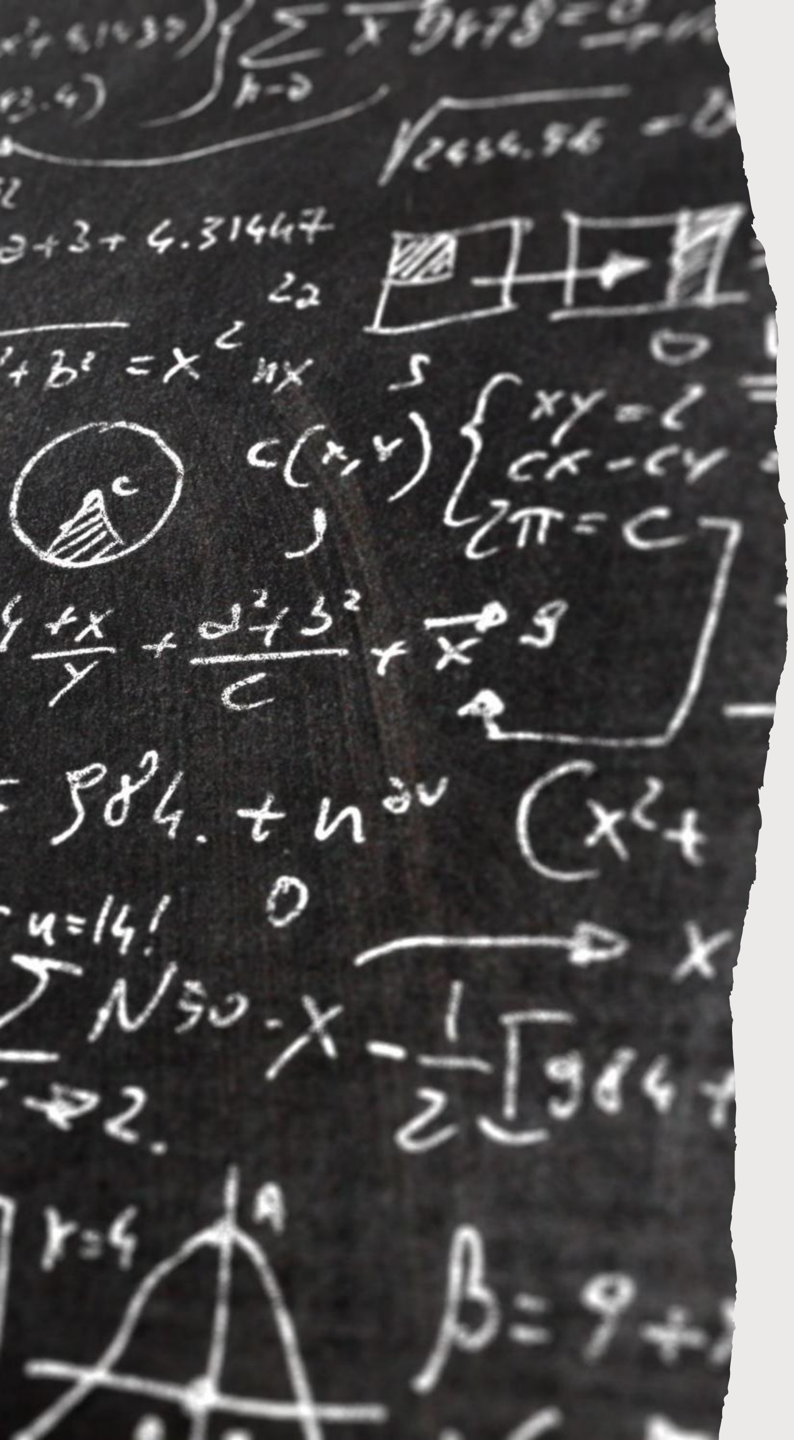
Country*: Canada
Postal Code*: T5J1X1
Phone Number*: 7804444444
Email*: ssmith@hotmail.com
Customer Status*: Bronze

Form Actions (Buttons) & Feedback

CustomerEdit.razor.cs



```
        //  update that data has change
        StateHasChanged();
    }
    catch (Exception ex)
    {
        // capture any exception message for display
        errorMessage = ex.Message;
    }
}

// save the customer
1 reference
private void Save()
{

}

//  Cancels/closes this instance.
1 reference
private void Cancel()
{

}

#endregion
```



Create “Save()“Method in LINQPad

- Copy the “Library->AddEditCustomer()” method to the “Customer Service.”
- Copy the “CodeBehind->AddEditCustomer()” method to the “CustomerEdit.razor.cs.”

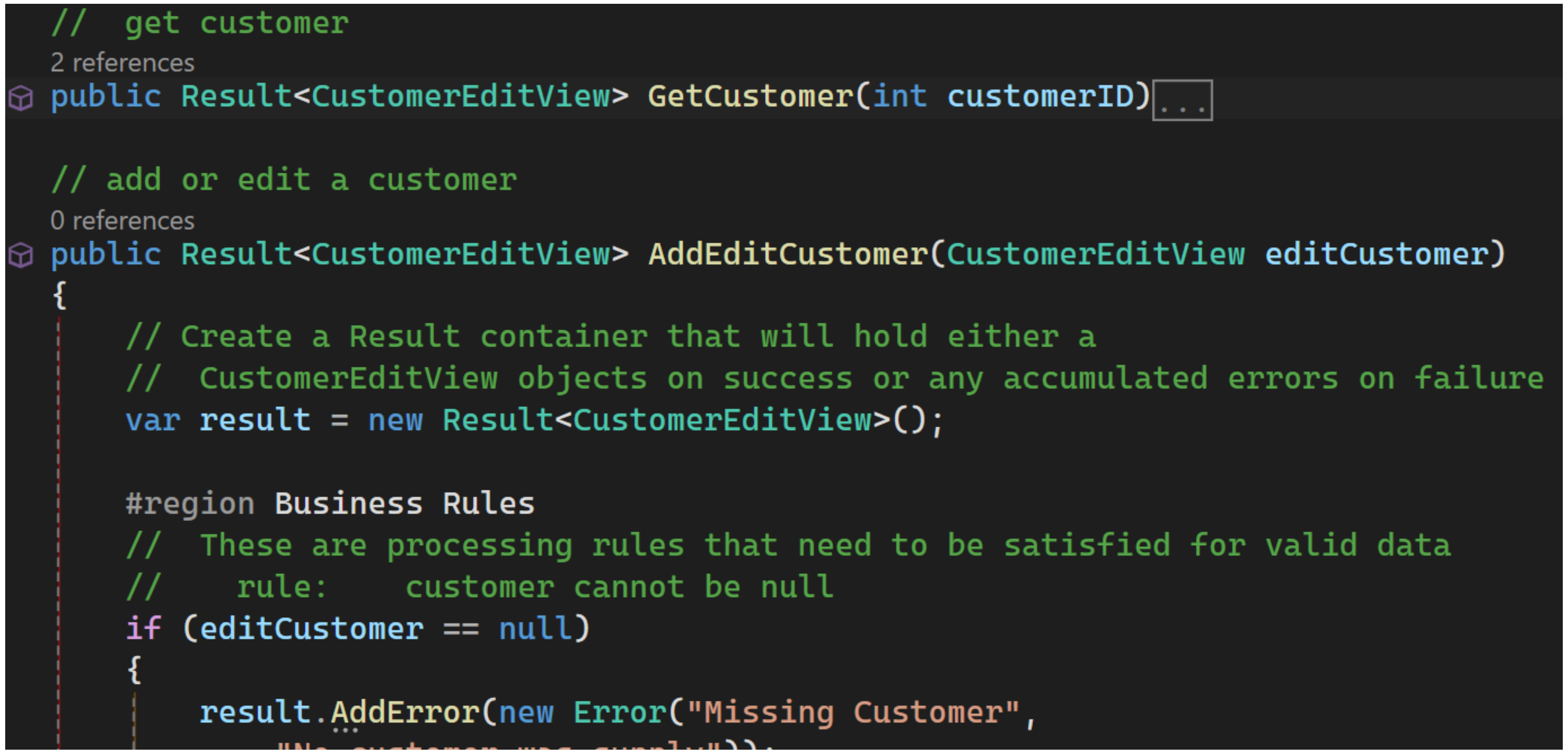
Code the "Save()"Method in LINQPad (Continue)

CustomerService.cs

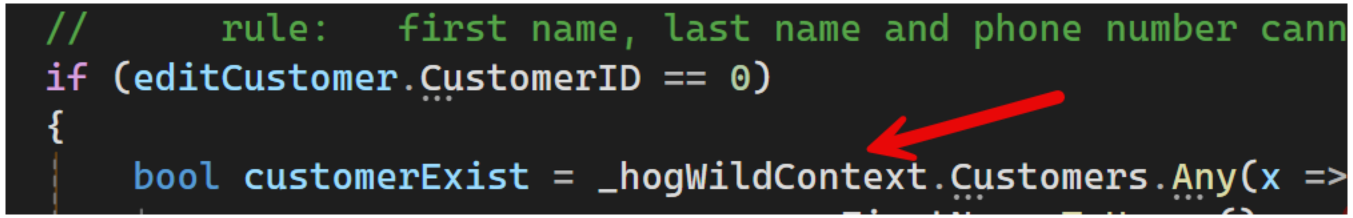
```
    if (errorList.Count > 0)
    {
        //  we need to clear the "track changes"
        //      otherwise we leave our entity system in flux
        ChangeTracker.Clear();
        //  throw the list of business processing error(s)
        throw new AggregateException(
            "Unable to add or edit customer. Please check error message(s)",
            errorList);
    }
    else
    {
        //  new customer
        if (customer.CustomerID == 0)
            Customers.Add(customer);
        else
            Customers.Update(customer);
        SaveChanges();
    }
    //  can return current editCustomer
    editCustomer.CustomerID = customer.CustomerID;
    return editCustomer;
}
```

Update Customer Service with "AddEditCustomer()" Method

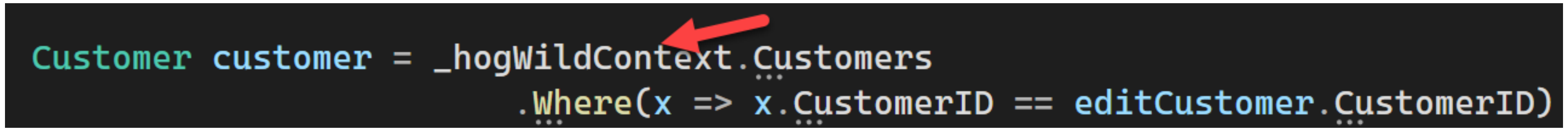
- Copy your method into the "Customer Service" class.
- Update all the customer entities references with the HogWildContext
- **NOTE: If you have followed the BYSResults pattern that we have created by building the functionality in LINQPad, then you do not have to do this step as we are already using the "_hogWildContext"**



```
//  get customer
2 references
public Result<CustomerEditView> GetCustomer(int customerID)...

// add or edit a customer
0 references
public Result<CustomerEditView> AddEditCustomer(CustomerEditView editCustomer)
{
    // Create a Result container that will hold either a
    //  CustomerEditView objects on success or any accumulated errors on failure
    var result = new Result<CustomerEditView>();

    #region Business Rules
    //  These are processing rules that need to be satisfied for valid data
    //    rule:    customer cannot be null
    if (editCustomer == null)
    {
        result.AddError(new Error("Missing Customer",
            "No customer was supplied"));
```



```
//      rule:   first name, last name and phone number cann
if (editCustomer.CustomerID == 0)
{
    bool customerExist = _hogWildContext.Customers.Any(x =>
```

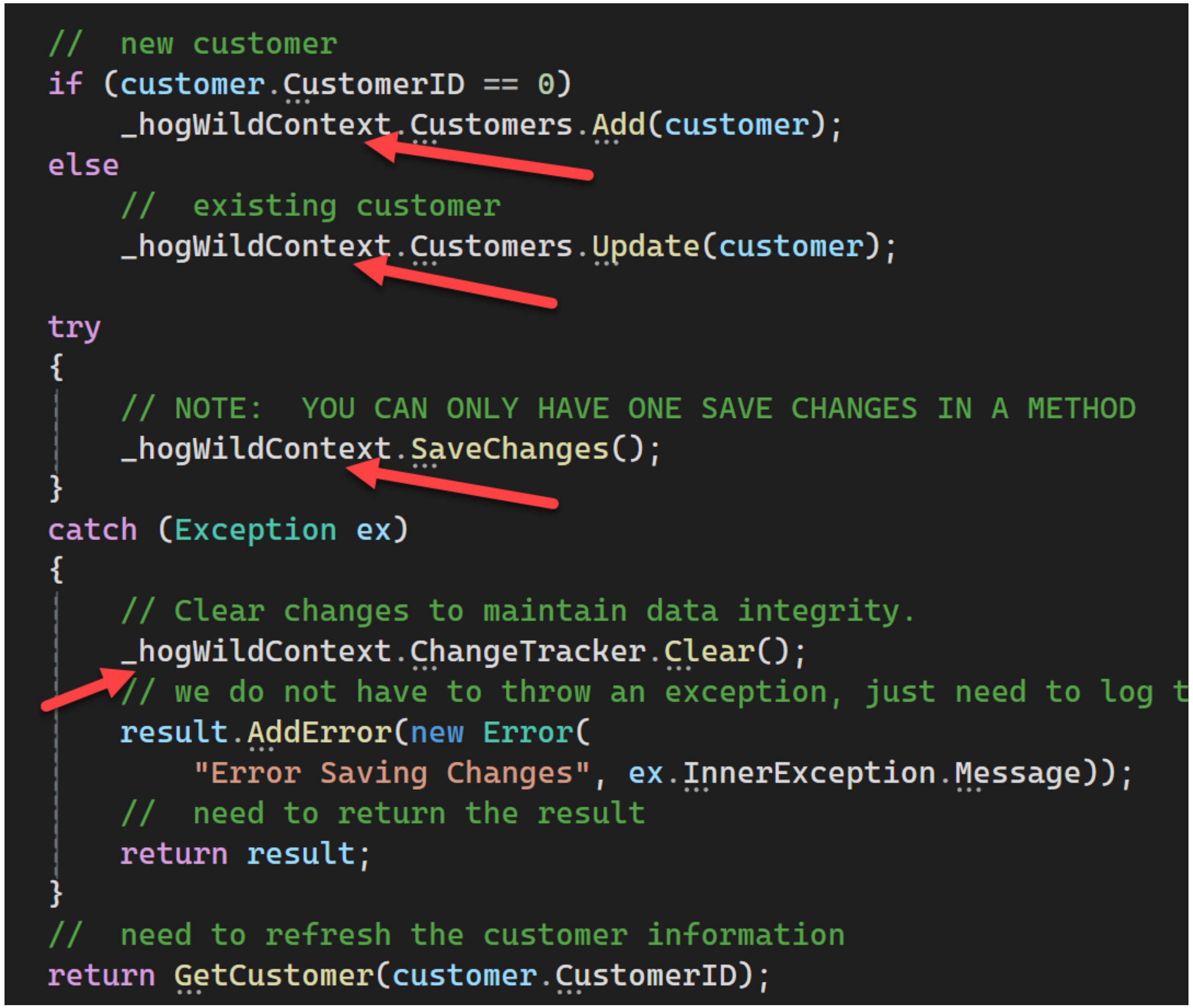


```
Customer customer = _hogWildContext.Customers
                        .Where(x => x.CustomerID == editCustomer.CustomerID)
```

CustomerService.cs

Update Customer Service with "AddEditCustomer()" Method

- Update all the customer entities references with the HogWildContext
- **NOTE: If you have followed the BYSResults pattern that we have created by building the functionality in LINQPad, then you do not have to do this step as we are already using the "_hogWildContext"**



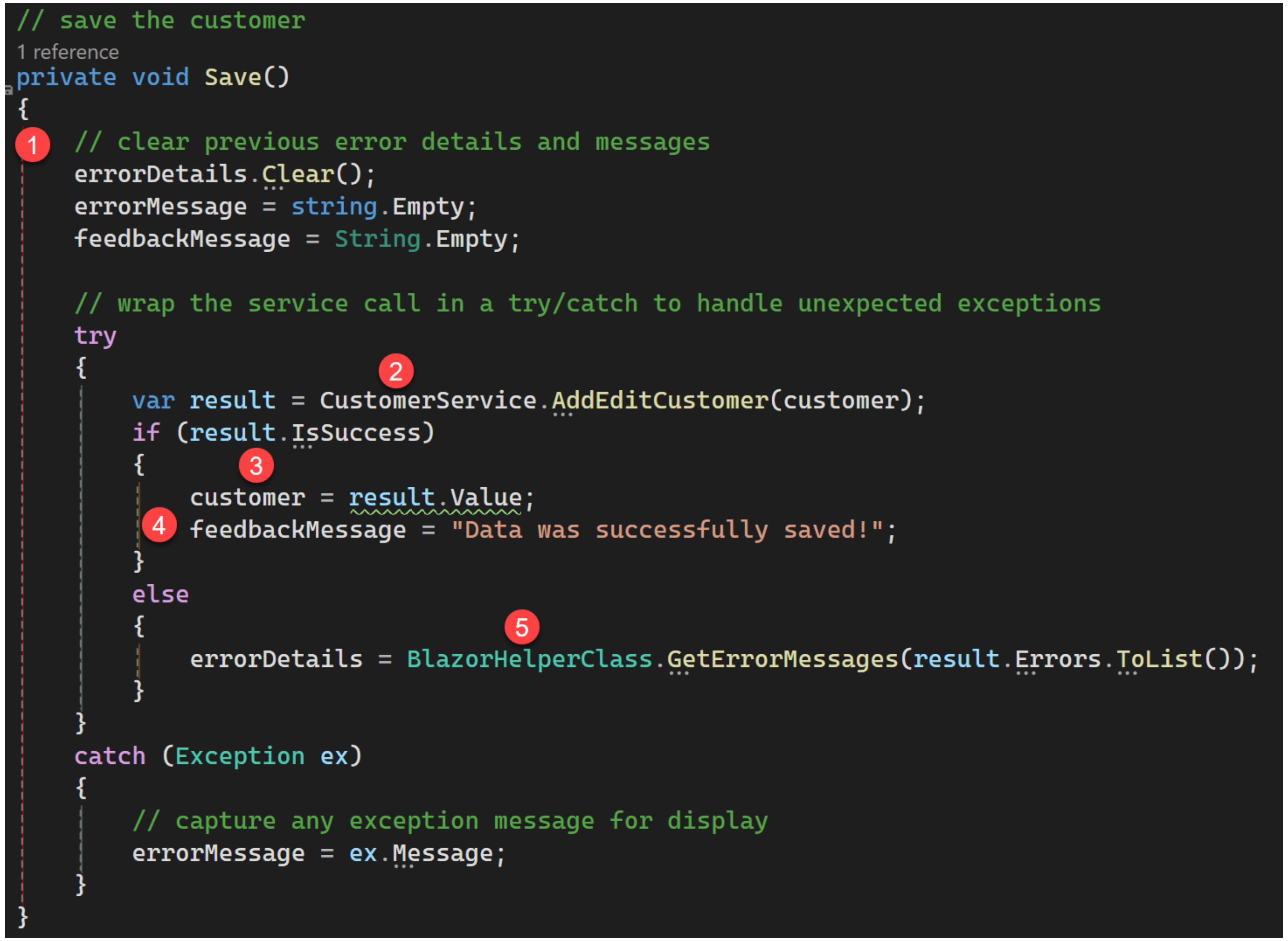
```
//  new customer
if (customer.CustomerID == 0)
    _hogWildContext.Customers.Add(customer);
else
    //  existing customer
    _hogWildContext.Customers.Update(customer);

try
{
    // NOTE:  YOU CAN ONLY HAVE ONE SAVE CHANGES IN A METHOD
    _hogWildContext.SaveChanges();
}
catch (Exception ex)
{
    // Clear changes to maintain data integrity.
    _hogWildContext.ChangeTracker.Clear();
    // we do not have to throw an exception, just need to log t
    result.AddError(new Error(
        "Error Saving Changes", ex.InnerException.Message));
    //  need to return the result
    return result;
}
//  need to refresh the customer information
return GetCustomer(customer.CustomerID);
```

CustomerService.cs

Update Razor Save Method

CustomerEdit.razor.cs

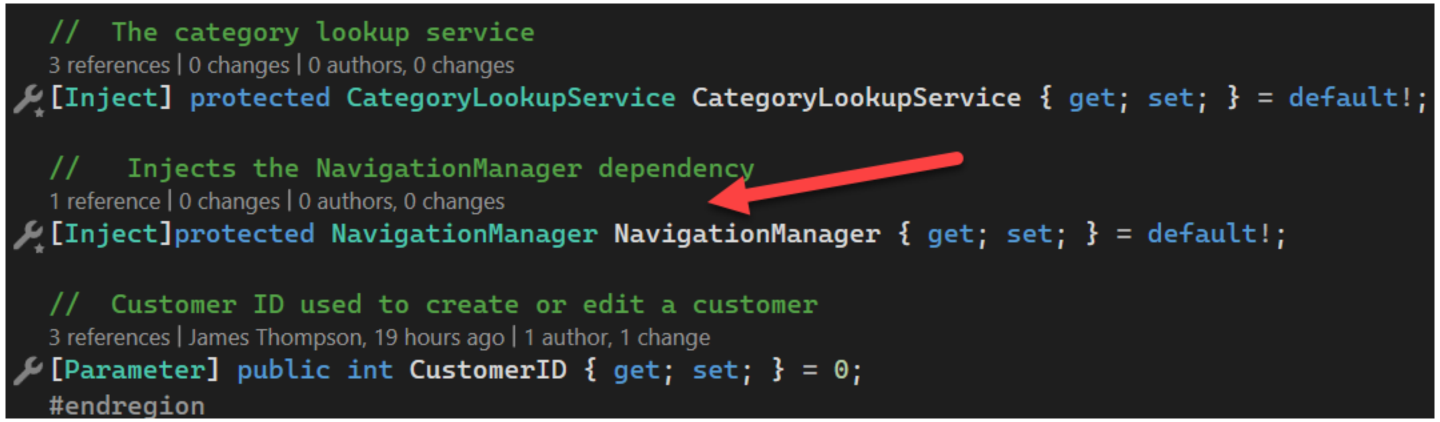
1. Copy the code from GetCustomer.linq within your CodeBehind -> AddEditCustomer
2. Updated the “service class” name to “CustomerService”
3. Add successful feedback
4. Rename “Customer” to “customer”
5. Add BlazorHelperClass to the GetErrorMessages



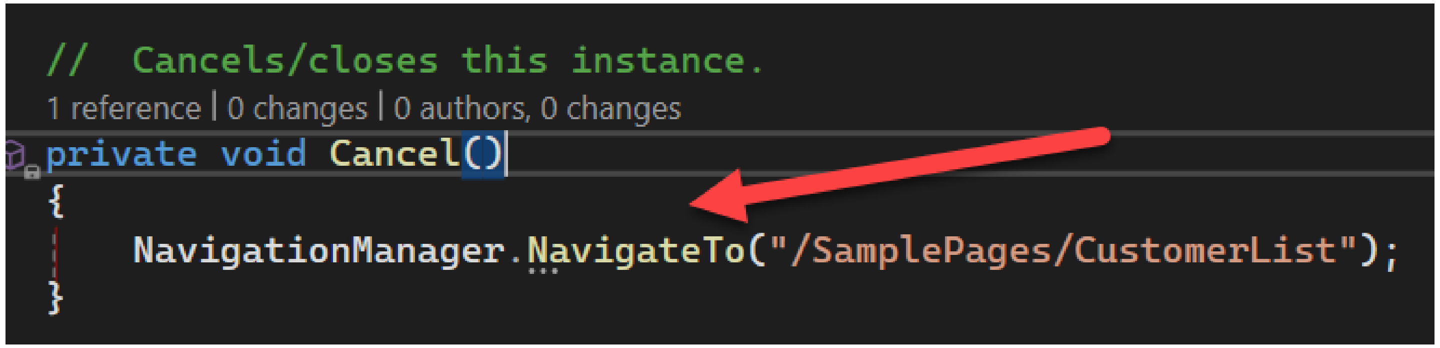
```
// save the customer
1 reference
private void Save()
{
    // clear previous error details and messages
    errorDetails.Clear();
    errorMessage = string.Empty;
    feedbackMessage = String.Empty;

    // wrap the service call in a try/catch to handle unexpected exceptions
    try
    {
        var result = CustomerService.AddEditCustomer(customer);
        if (result.IsSuccess)
        {
            customer = result.Value;
            feedbackMessage = "Data was successfully saved!";
        }
        else
        {
            errorDetails = BlazorHelperClass.GetErrorMessages(result.Errors.ToList());
        }
    }
    catch (Exception ex)
    {
        // capture any exception message for display
        errorMessage = ex.Message;
    }
}
```

Update Razor Cancel Method

CustomerEdit.razor.cs

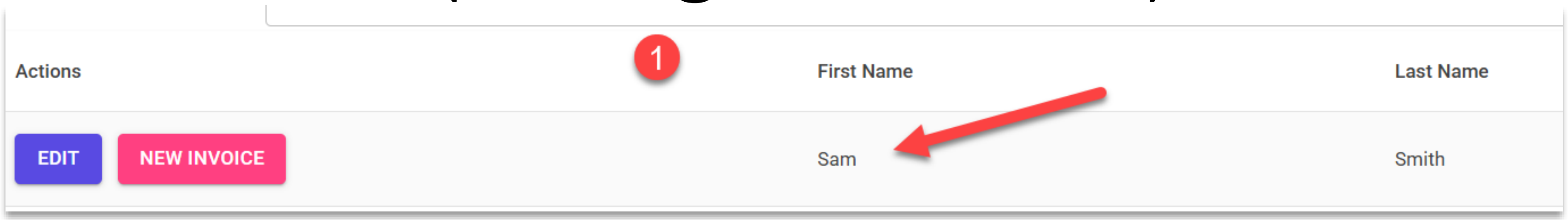


```
//  The category lookup service
3 references | 0 changes | 0 authors, 0 changes
[Inject] protected CategoryLookupService CategoryLookupService { get; set; } = default!;

//   Injects the NavigationManager dependency
1 reference | 0 changes | 0 authors, 0 changes
[Inject]protected NavigationManager NavigationManager { get; set; } = default!;

//  Customer ID used to create or edit a customer
3 references | James Thompson, 19 hours ago | 1 author, 1 change
[Parameter] public int CustomerID { get; set; } = 0;
#endregion
```



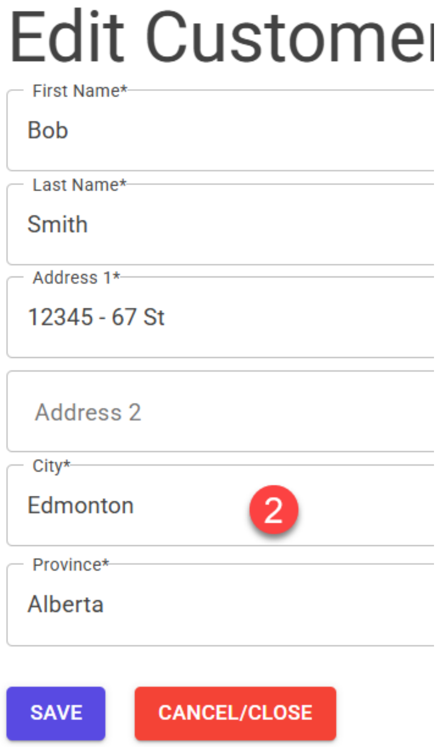
```
//  Cancels/closes this instance.
1 reference | 0 changes | 0 authors, 0 changes
private void Cancel()
{
    NavigationManager.NavigateTo("/SamplePages/CustomerList");
}
```

Save Data (Saving Valid Data)



1

Actions	First Name	Last Name
EDIT NEW INVOICE	Sam	Smith



Edit Custome

First Name*
Bob

Last Name*
Smith

Address 1*
12345 - 67 St

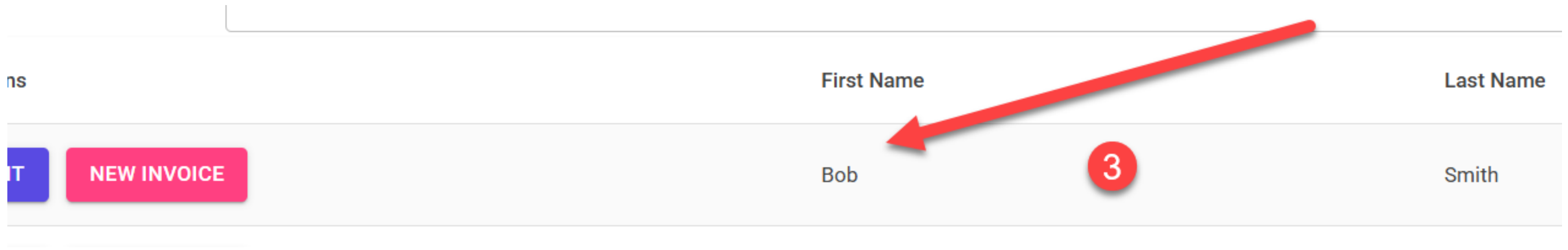
Address 2

City*
Edmonton

2

Province*
Alberta

SAVE CANCEL/CLOSE

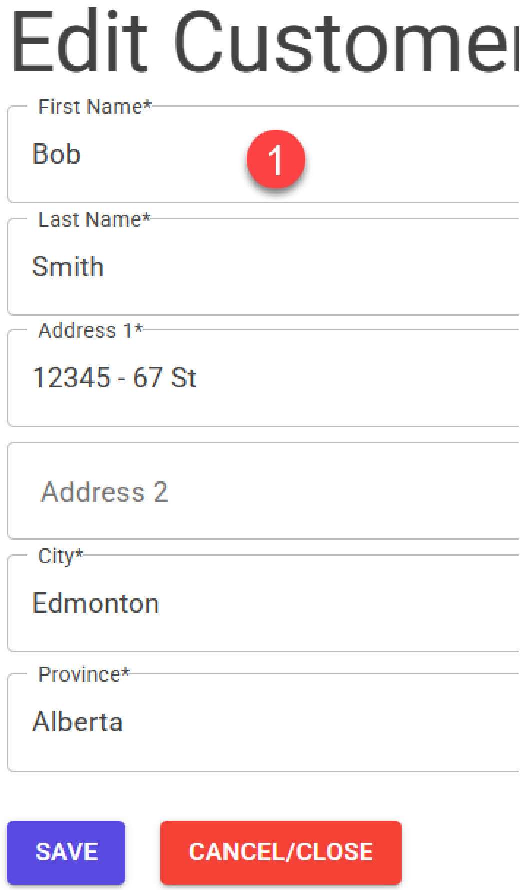


3

ns	First Name	Last Name
IT NEW INVOICE	Bob	Smith

Cancel Data

- On the “Customer Edit” page, if a user presses the “Cancel/Close” button, the page will be redirected (routed) back to the “Customer List” page.



Edit Custome

First Name*: Bob

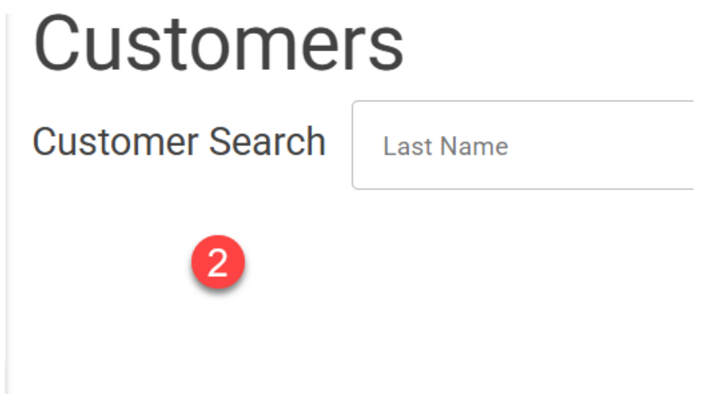
Last Name*: Smith

Address 1*: 12345 - 67 St

Address 2

City*: Edmonton

Province*: Alberta



Customers

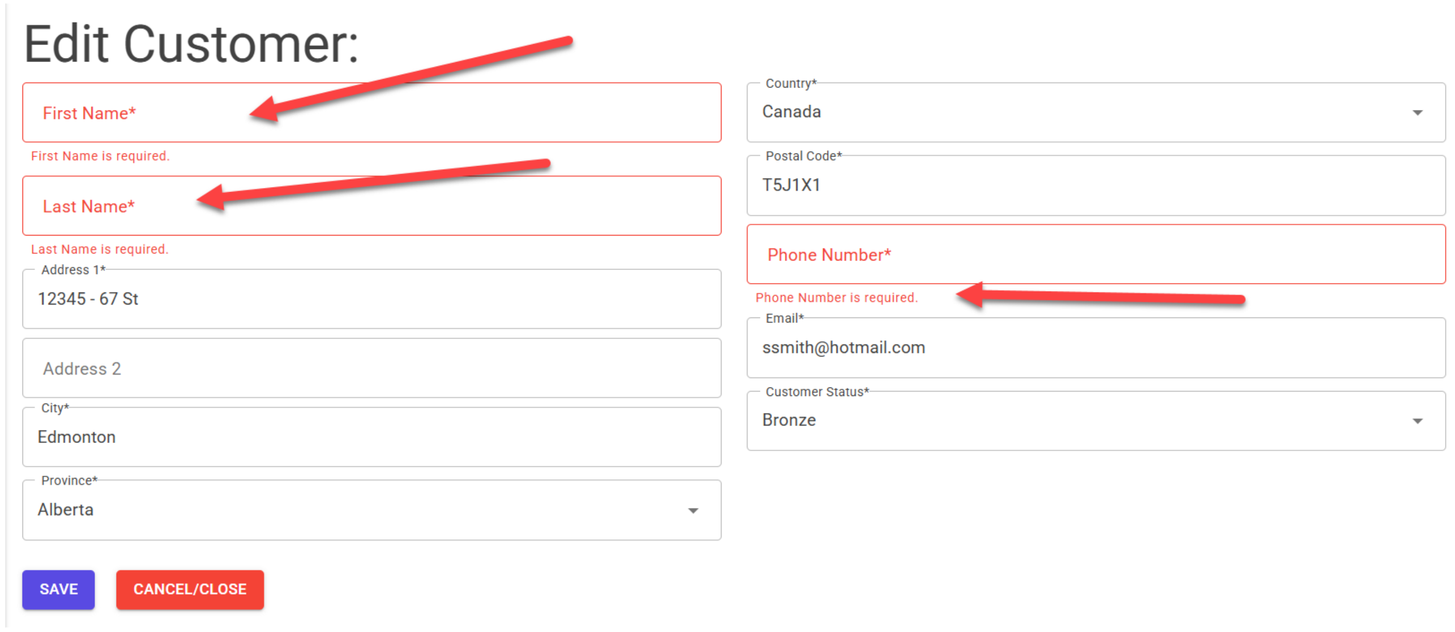
Customer Search: Last Name

Known Issues

- If the user presses “Save,” the web page will send the unmodified data to the server repeatedly.
- Editing a record and hitting “Close”, there is no feedback.

Saving Known Bad Data

- When saving required data, all validation is done on the server. Though this approach ensures protection from an injection, we can also stop the data from being sent if we know that the data is incorrect.



Edit Customer:

First Name*
First Name is required.

Last Name*
Last Name is required.

Address 1*: 12345 - 67 St

Address 2

City*: Edmonton

Province*: Alberta

Country*: Canada

Postal Code*: T5J1X1

Phone Number*
Phone Number is required.

Email*: ssmith@hotmail.com

Customer Status*: Bronze

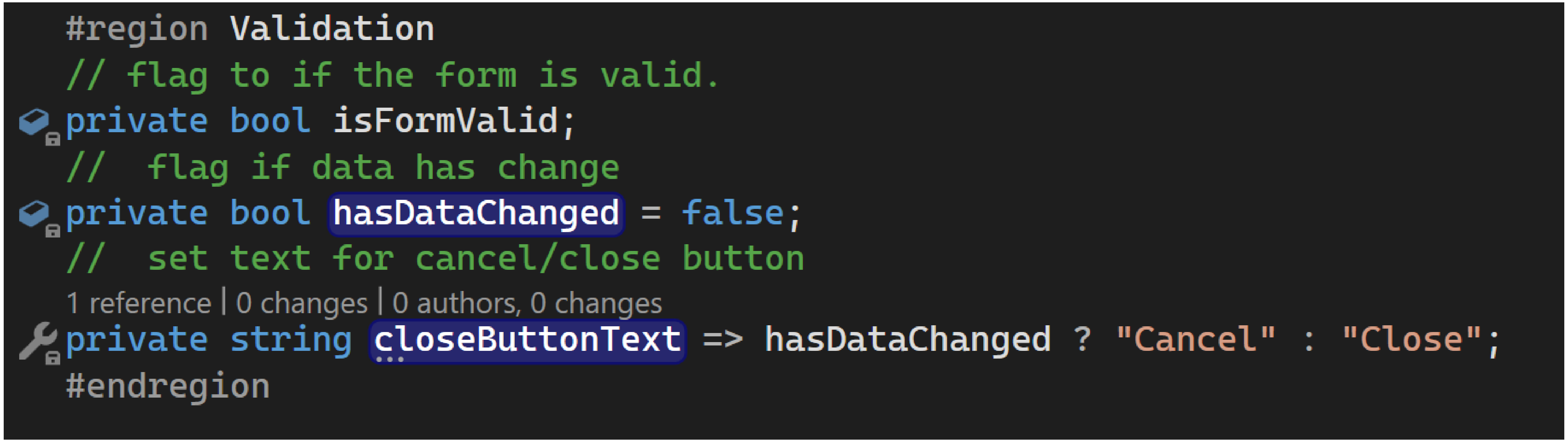
Controlling Save/Cancel Behavior

1. Add the **"Is Form Valid"** property.
2. Add the **“Has Data Change"** property.
3. Dynamically disable the **Save** button when the data is either invalid or unchanged.
4. Dynamically update the text on the **Close/Cancel** button depending on whether the data has changed.

Add Validation Code

- Add Validation Region
- Add isFormValid
- Add hasDataChanged
- Add closeButtonText

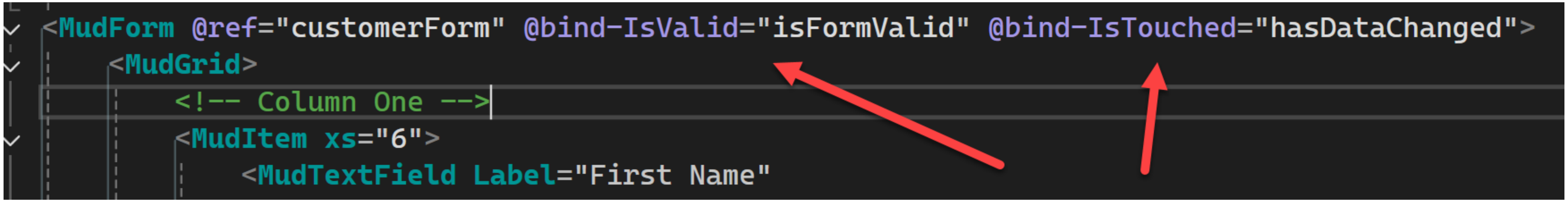
CustomerEdit.razor.cs



```
#region Validation
// flag to if the form is valid.
private bool isFormValid;
//  flag if data has change
private bool hasDataChanged = false;
//  set text for cancel/close button
1 reference | 0 changes | 0 authors, 0 changes
private string closeButtonText => hasDataChanged ? "Cancel" : "Close";
#endregion
```

Add Validation Code (Continue)

- Add isFormValid
- Add hasDataChanged
- @bind-IsTouched in **MudBlazor's MudForm** automatically tracks whether any input field in the form has been modified, updating the bound boolean variable (true if a change is made, false after a reset).



```
<MudForm @ref="customerForm" @bind-IsValid="isFormValid" @bind-IsTouched="hasDataChanged">
    <MudGrid>
        <!-- Column One -->
        <MudItem xs="6">
            <MudTextField Label="First Name"
```

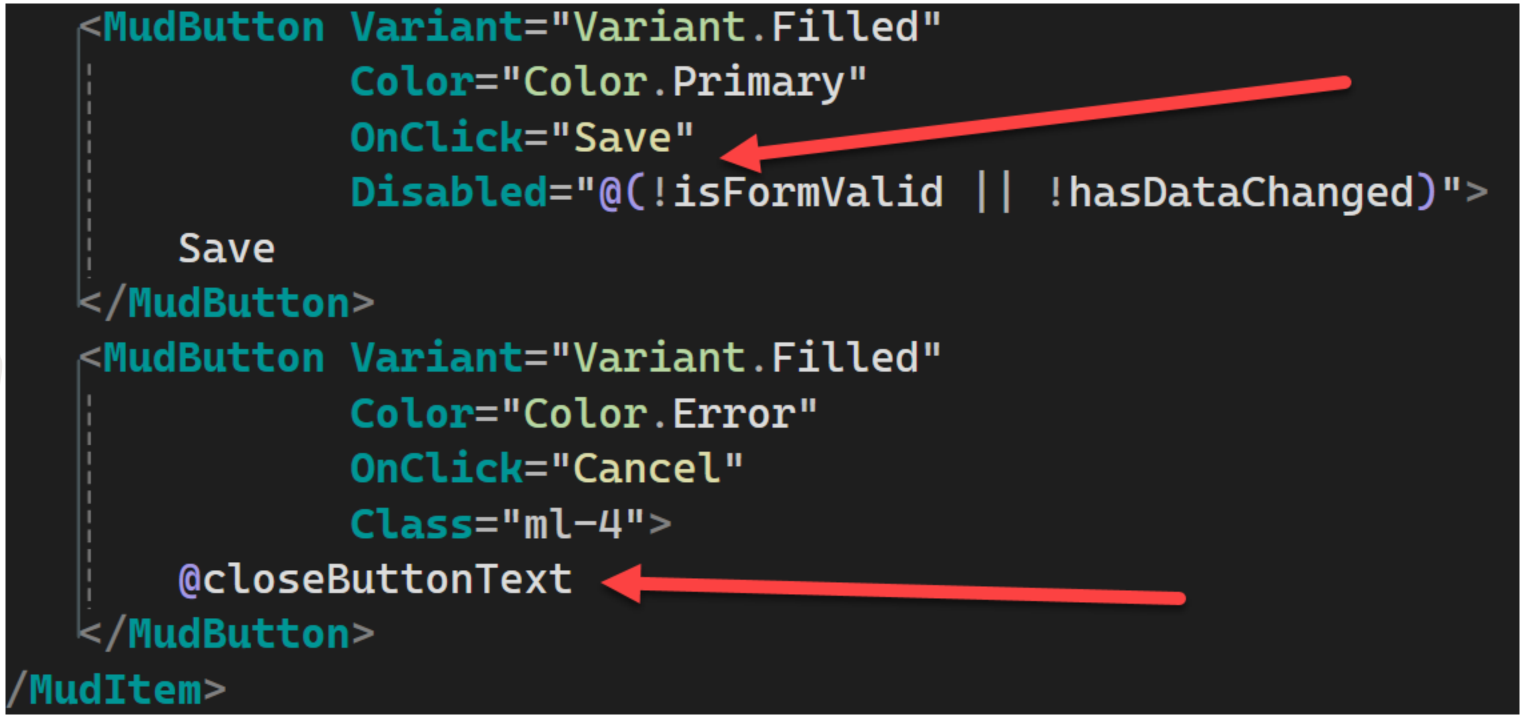
CustomerEdit.razor

Refactor / Update Buttons

The **Save** button is disabled if the form data is invalid or has not been modified

The **Cancel** button text updates when there are unsaved changes.

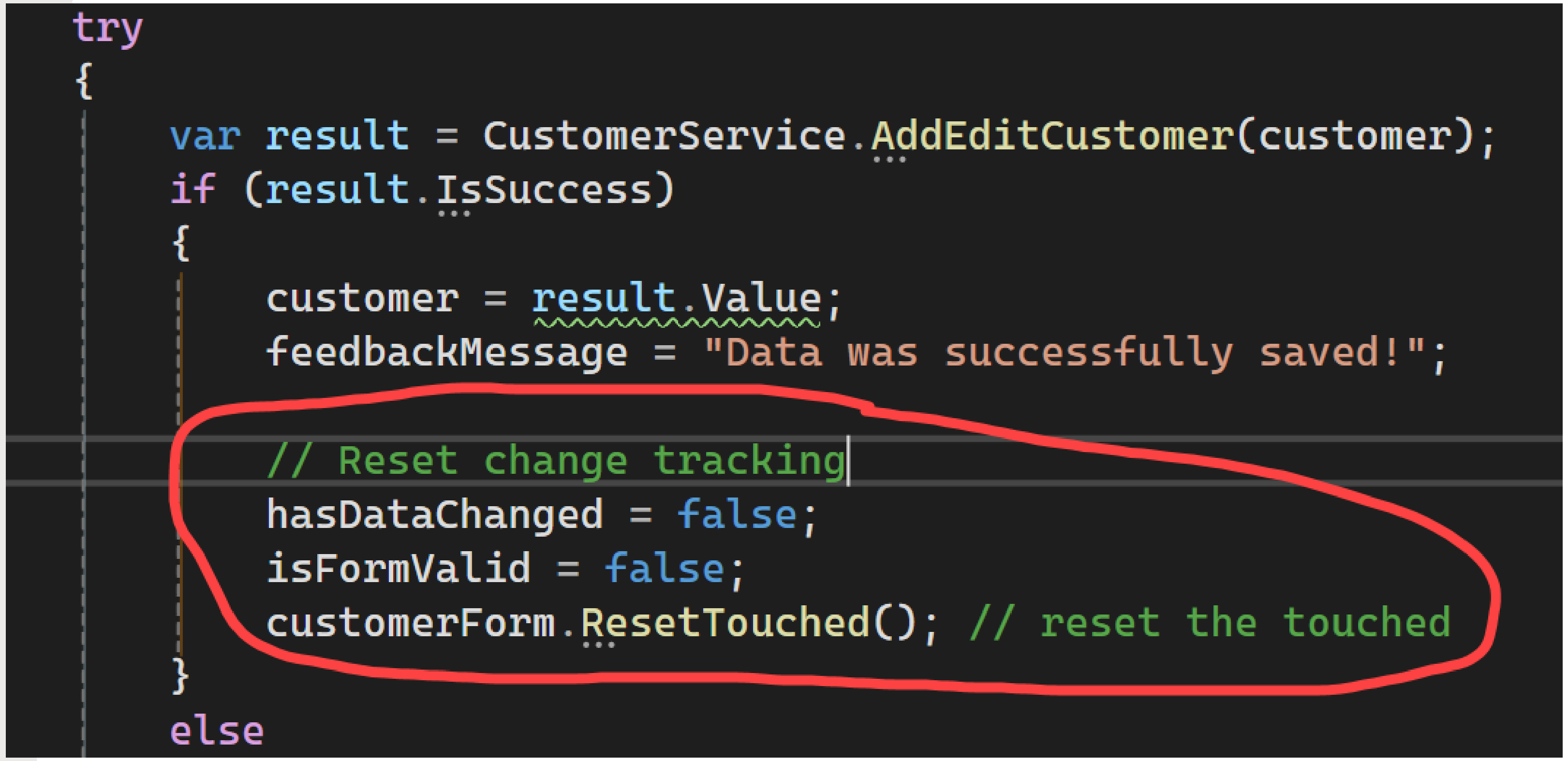
CustomerEdit.razor



```
    <MudButton Variant="Variant.Filled"
               Color="Color.Primary"
               OnClick="Save"
               Disabled="@(!isFormValid || !hasDataChanged)">
        Save
    </MudButton>
    <MudButton Variant="Variant.Filled"
               Color="Color.Error"
               OnClick="Cancel"
               Class="ml-4">
        @closeButtonText
    </MudButton>
/MudItem>
```

Refactor Save Method

- Resets the change tracking flag, indicating that no modifications have been made to the form.
- Marks the form as invalid, requiring re-validation before enabling the save.
- Resets the form's touched state, clearing any modification tracking for input fields.

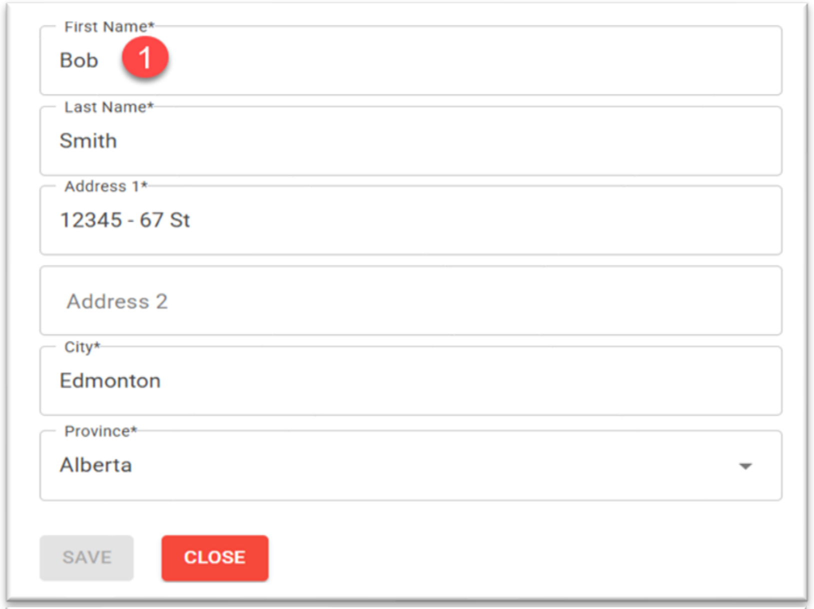


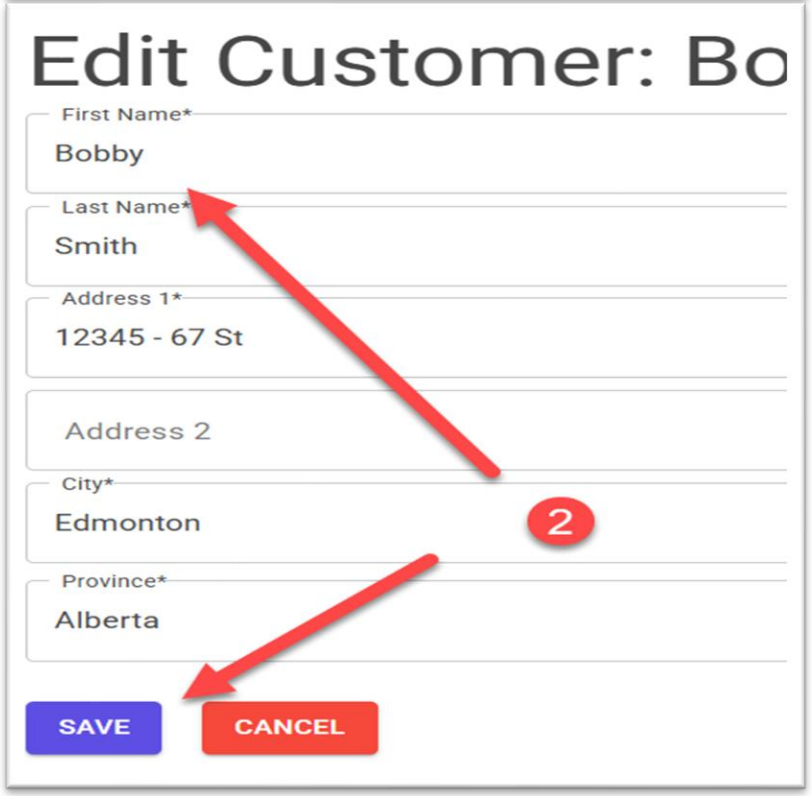
```
try
{
    var result = CustomerService.AddEditCustomer(customer);
    if (result.IsSuccess)
    {
        customer = result.Value;
        feedbackMessage = "Data was successfully saved!";

        // Reset change tracking
        hasDataChanged = false;
        isFormValid = false;
        customerForm.ResetTouched(); // reset the touched
    }
    else
```

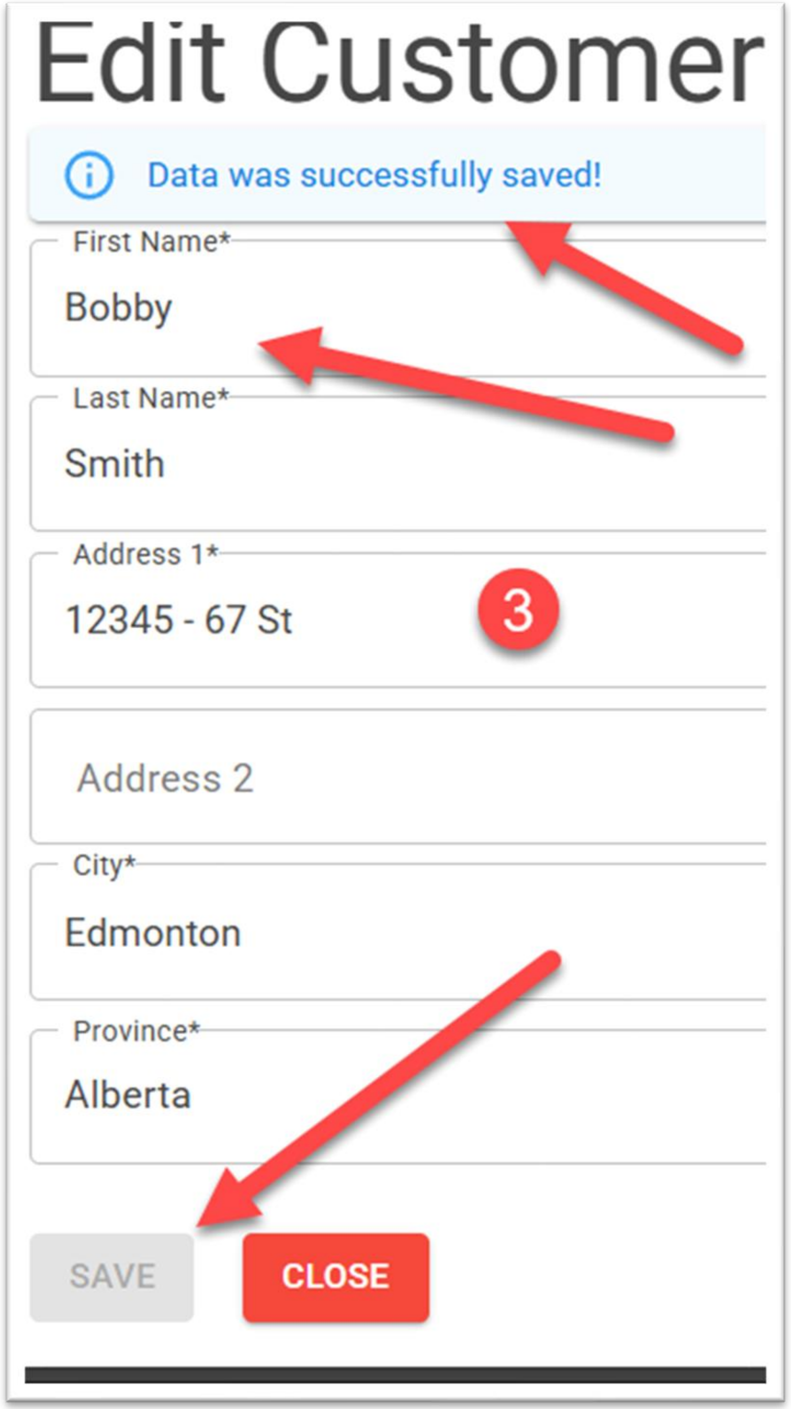
CustomerEdit.razor.cs

Blazor Page Outputs

1) Before editing customer.
2) Editing customer.
   - Save is Enabled.
   - Close/Cancel text changes.
3) After saving customer data.
   - Save button disable.
   - Close/Cancel text changes.

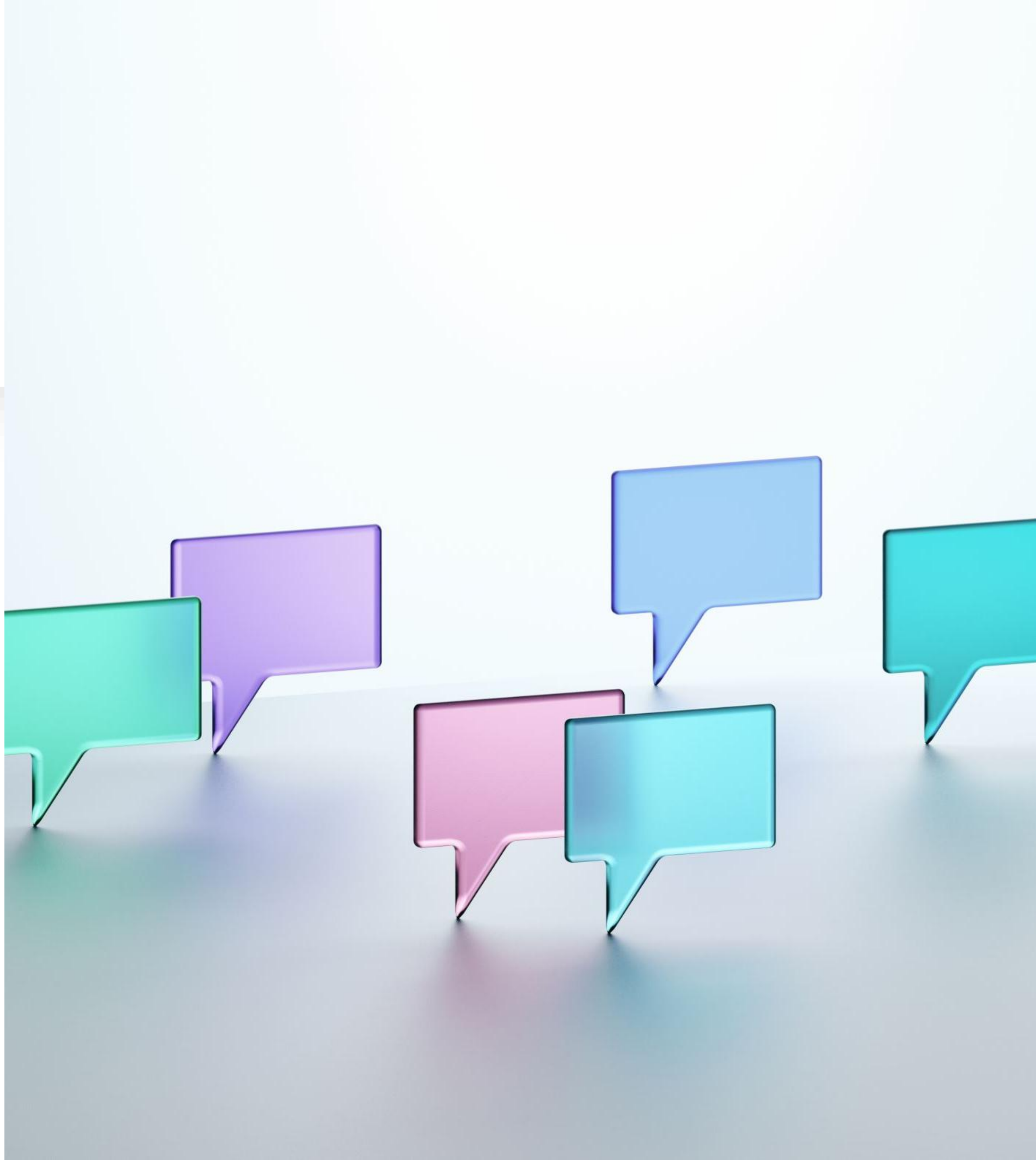

First Name*: Bob
Last Name*: Smith
Address 1*: 12345 - 67 St
Address 2
City*: Edmonton
Province*: Alberta


Edit Customer: Bo
First Name*: Bobby
Last Name*: Smith
Address 1*: 12345 - 67 St
Address 2
City*: Edmonton
Province*: Alberta


Edit Customer
Data was successfully saved!
First Name*: Bobby
Last Name*: Smith
Address 1*: 12345 - 67 St
Address 2
City*: Edmonton
Province*: Alberta

Providing Feedback on Cancelling Unsaved Changes (NOT COMPLETED)

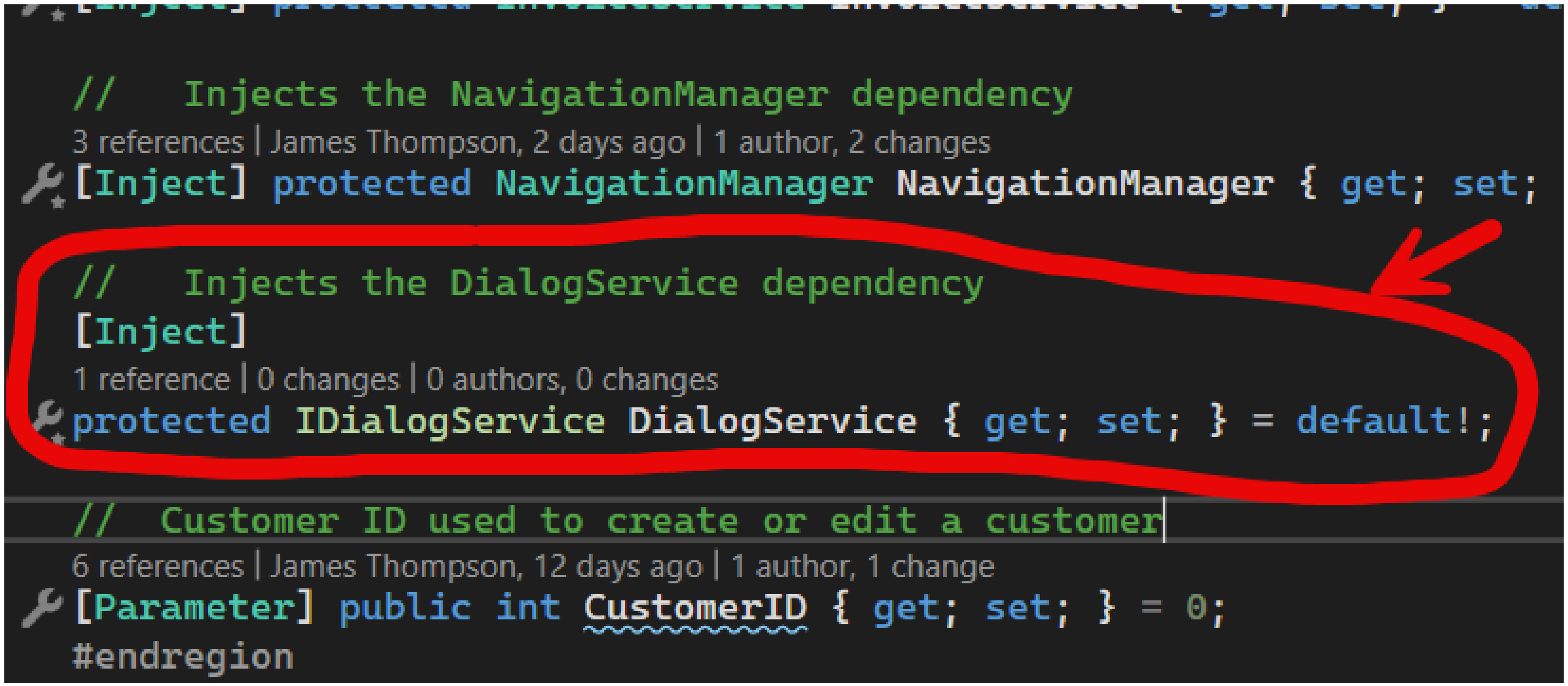
- When a user is editing a customer, there is no confirmation prompt before discarding unsaved changes.
- We will provide a dialog when there are unsaved changes.
- NOTE: We will use the “DialogService” component.



DialogService

CustomerEdit.razor.cs

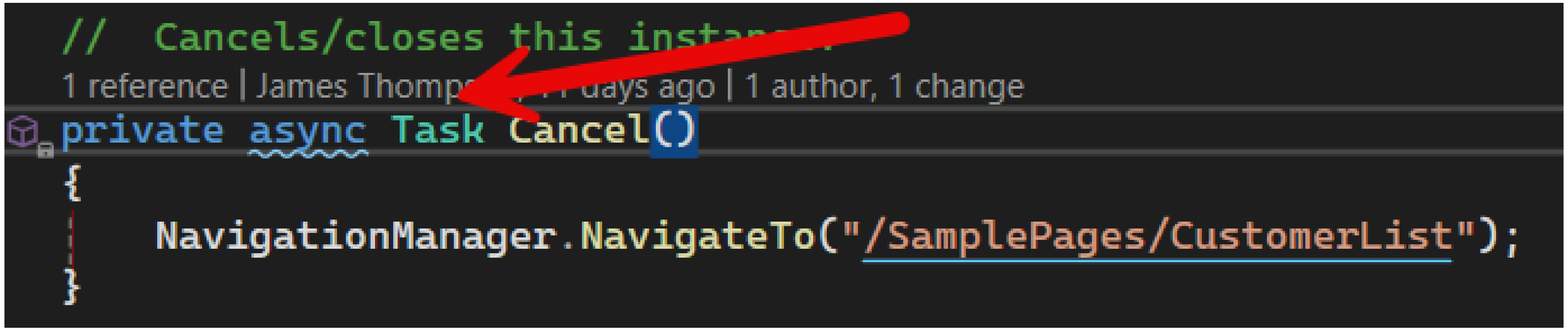
- Enables components to prompt users with confirmation dialogs, alerts, and other interactive models, enhancing user experience and decision-making processes.



```
//   Injects the NavigationManager dependency
3 references | James Thompson, 2 days ago | 1 author, 2 changes
[Inject] protected NavigationManager NavigationManager { get; set;

//   Injects the DialogService dependency
[Inject]
1 reference | 0 changes | 0 authors, 0 changes
protected IDialogService DialogService { get; set; } = default!;

//  Customer ID used to create or edit a customer
6 references | James Thompson, 12 days ago | 1 author, 1 change
[Parameter] public int CustomerID { get; set; } = 0;
#endregion
```

Refactor the Cancel Method

- Change the “Cancel” return type from void to “async Task”

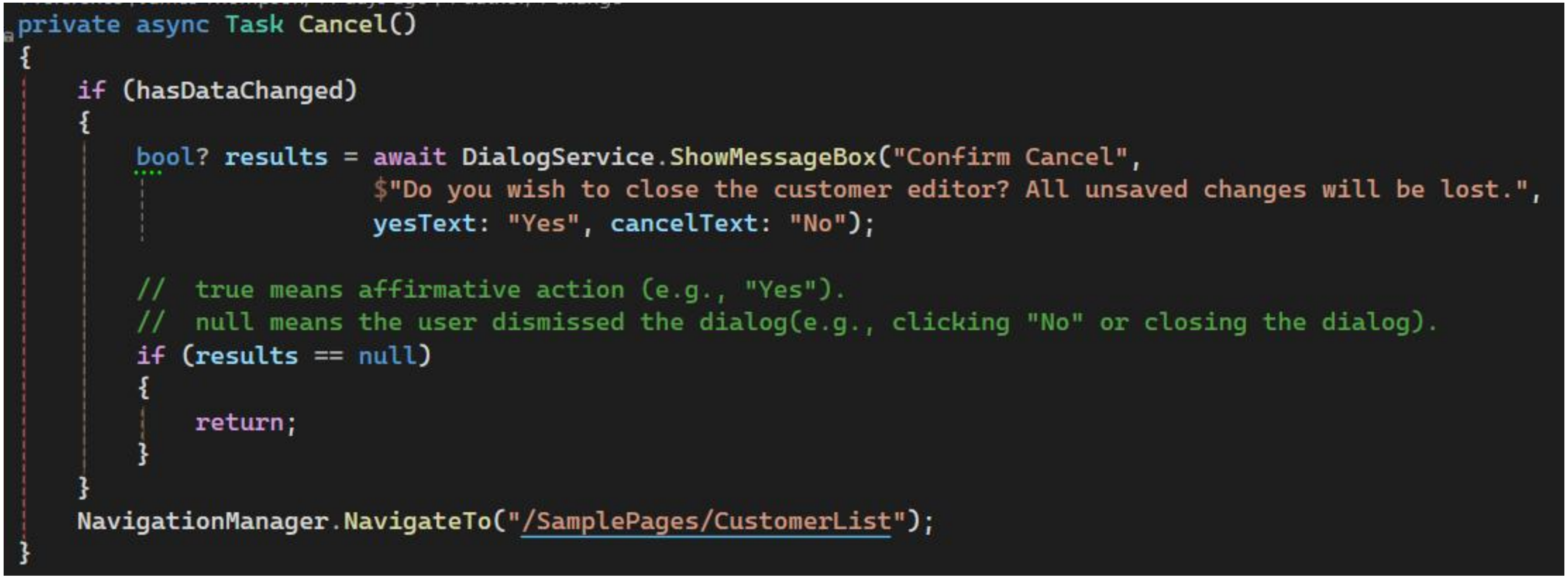


```
// Cancels/closes this instance.
1 reference | James Thompson | ... days ago | 1 author, 1 change
private async Task Cancel()
{
    NavigationManager.NavigateTo("/SamplePages/CustomerList");
}
```

CustomerEdit.razor.cs

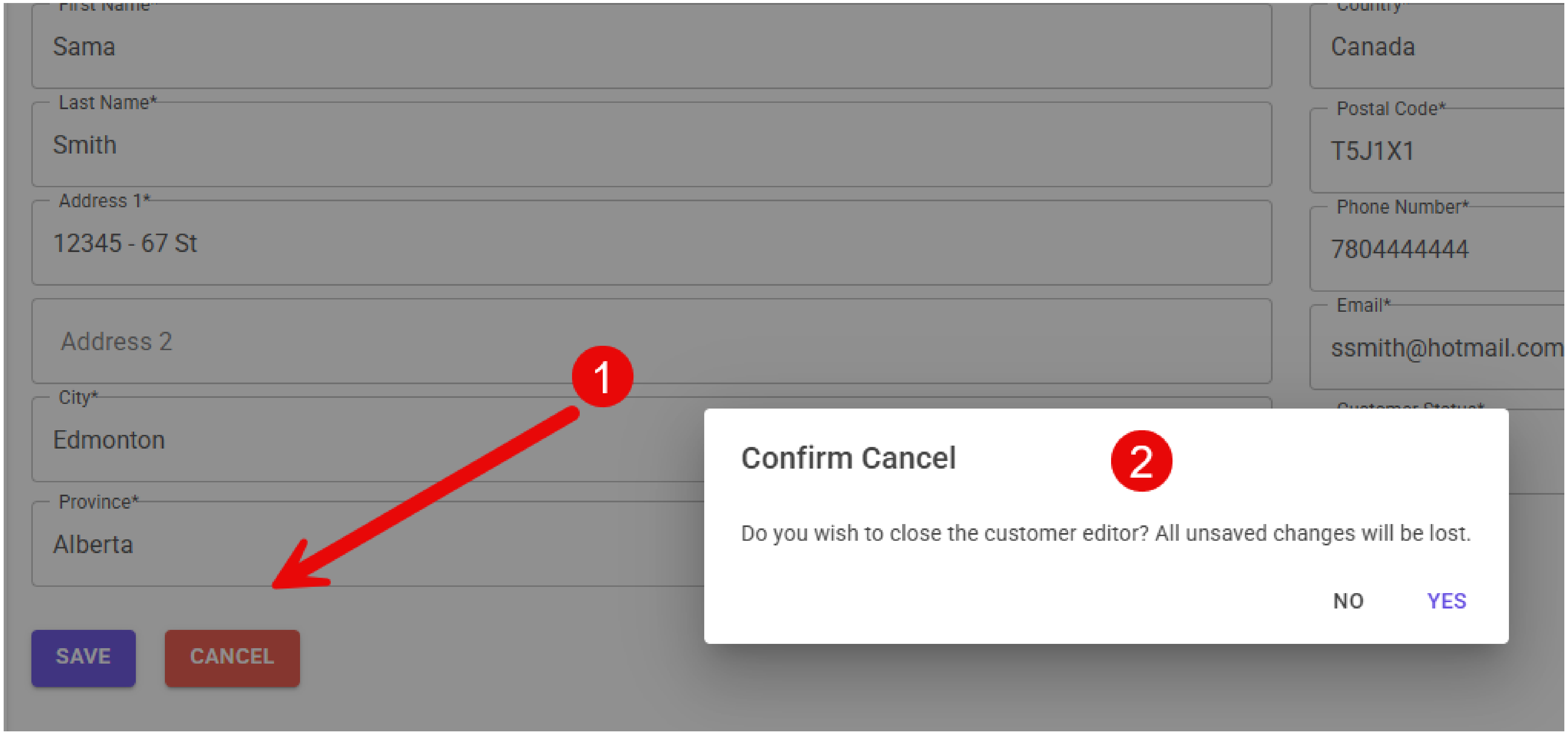
Refactor the Cancel Method (continue)

- Check to see if the data has changed
  - True
    - Prompt the user if they wish to discard unsaved changes
  - False
    - No need to prompt and return to the customer list



```
private async Task Cancel()
{
    if (hasDataChanged)
    {
        bool? results = await DialogService.ShowMessageBox("Confirm Cancel",
                        $"Do you wish to close the customer editor? All unsaved changes will be lost.",
                        yesText: "Yes", cancelText: "No");

        //  true means affirmative action (e.g., "Yes").
        //  null means the user dismissed the dialog(e.g., clicking "No" or closing the dialog).
        if (results == null)
        {
            return;
        }
    }
    NavigationManager.NavigateTo("/SamplePages/CustomerList");
}
```

Confirmation Dialog



First Name*
Sama

Last Name*
Smith

Address 1*
12345 - 67 St

Address 2

City*
Edmonton

Province*
Alberta

Country*
Canada

Postal Code*
T5J1X1

Phone Number*
7804444444

Email*
ssmith@hotmail.com

Customer Status*

Confirm Cancel

Do you wish to close the customer editor? All unsaved changes will be lost.

Statement Regarding Slide Accuracy and Potential Revisions

- Please note that the content of these PowerPoint slides is accurate to the best of my knowledge at the time of presentation. However, as new research and developments emerge, or to enhance the learning experience, these slides may be subject to updates or revisions in the future. I encourage you to stay engaged with the course materials and any announcements for the most current information