CST-350 Activity 6 Partial View

Table of Contents

Learning Goals

By the end of this lesson, you will be able to:

1. **Explain the Benefits of Partial Views**: Learn what partial views are and why they are useful in an ASP.NET web application.
2. **Create and Use Partial Views**: Know how to create partial views and use them within your main views.
3. **Handle AJAX Requests**: Learn how to use AJAX to load partial views dynamically, enhancing the user experience by updating parts of the page without a full page reload.

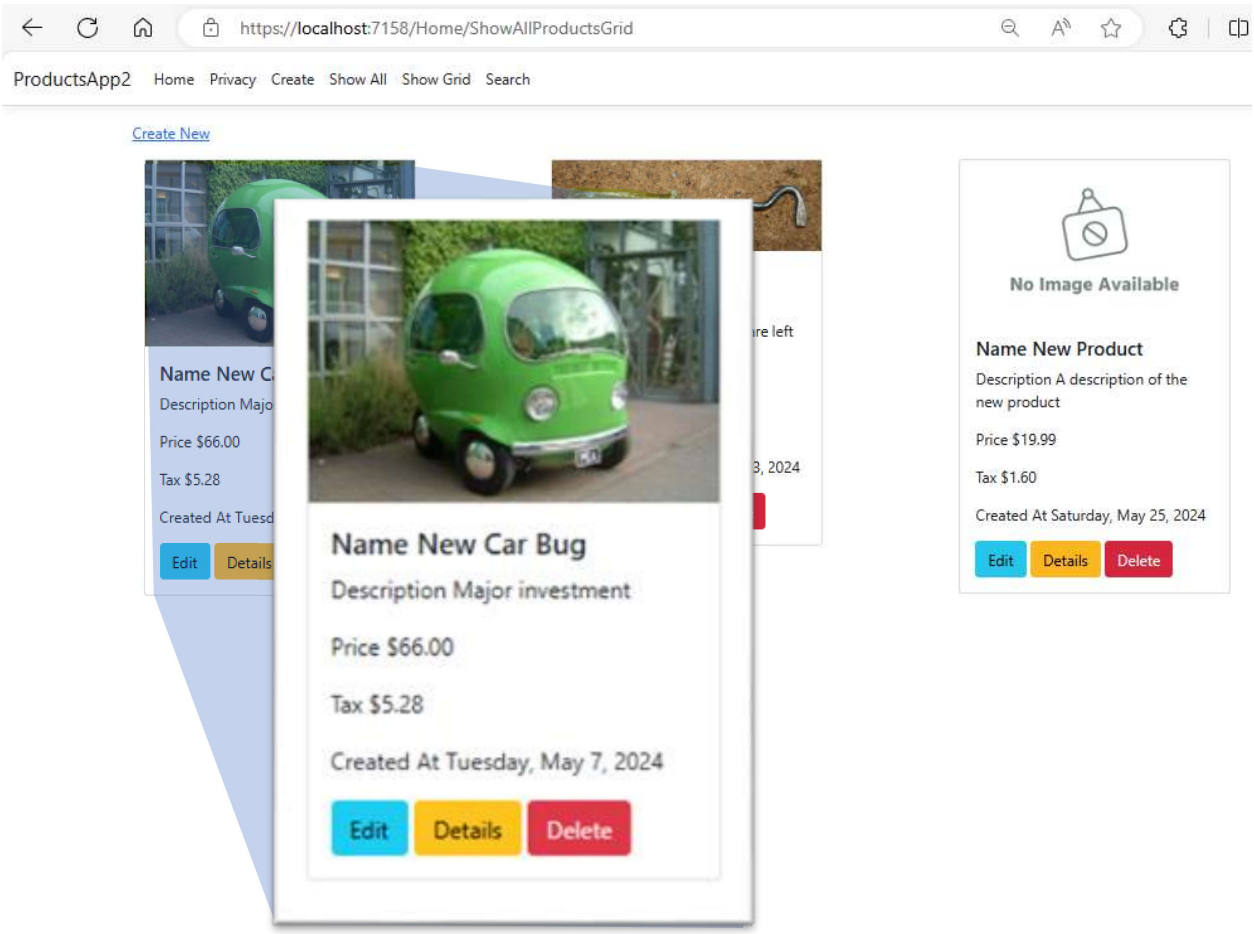


Figure 1 A single card (Partial Page) will be updated without refreshing the the entire page.

Key Terminology

- **Partial View**: A Razor view that renders a portion of content. It is reusable and can be embedded within other views.
- **AJAX (Asynchronous JavaScript and XML)**: A technique for creating fast and dynamic web pages. AJAX allows parts of a web page to be updated asynchronously by exchanging small amounts of data with the server behind the scenes.

Practical Example

In this lesson, we will work with two projects: ProductApp and ButtonGrid.

1. **ProductApp**:
    - We will create a partial view for displaying product details (**_ProductCard**) and use it in the **ShowAllProductsGrid** view.
    - We will implement a modal for editing products using AJAX to dynamically update product details without refreshing the page.
2. **ButtonGrid**:
    - We will refactor the **Index** view to use a partial view (**_Button**) for rendering each game button.

- We will handle button click events using AJAX to update the button state and image dynamically.

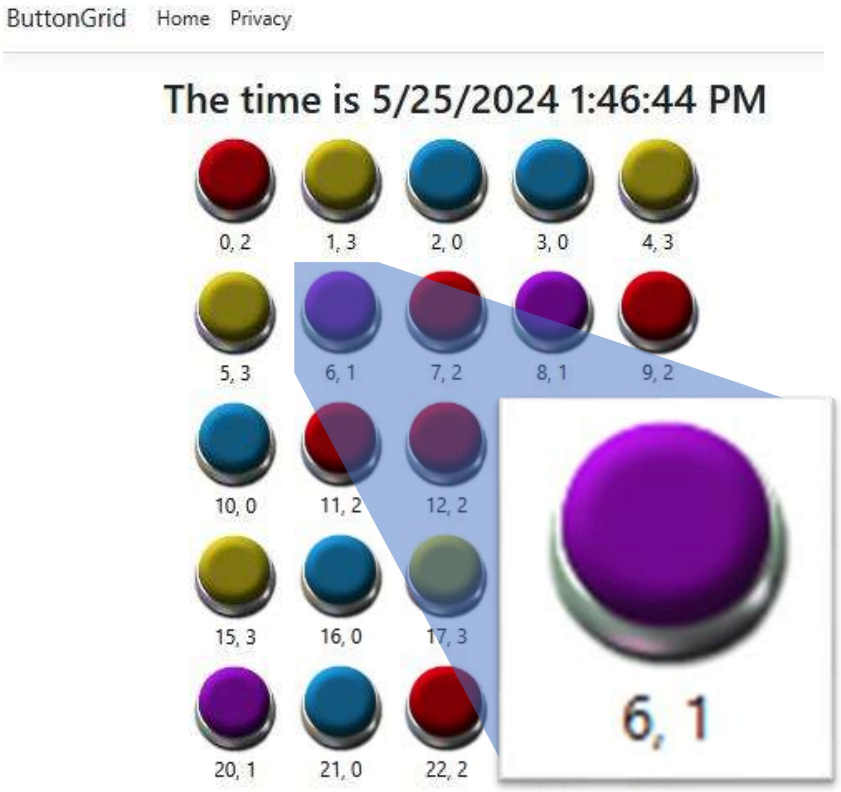


Figure 2 A single button (Partial Page) will be updated without reloading the entire page.

Part 1 - Partial Views in the Product App

1. Open the ProductApp created in a previous lesson.

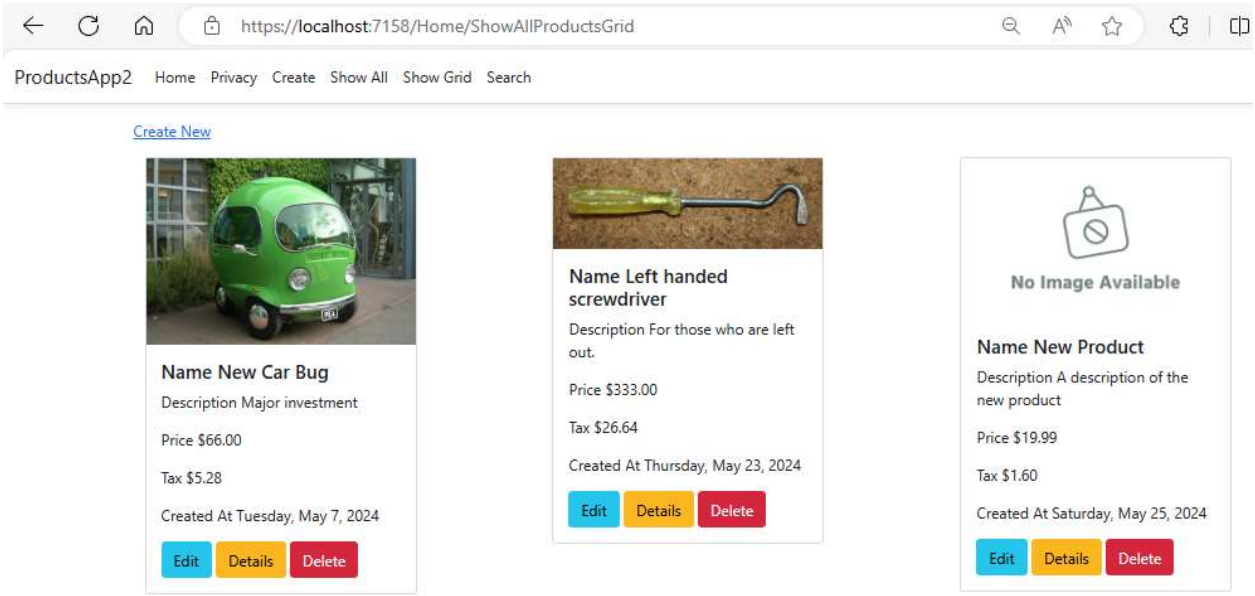


Figure 3 The "Grid View" of the Products App

2. Create a new View in the Home folder.

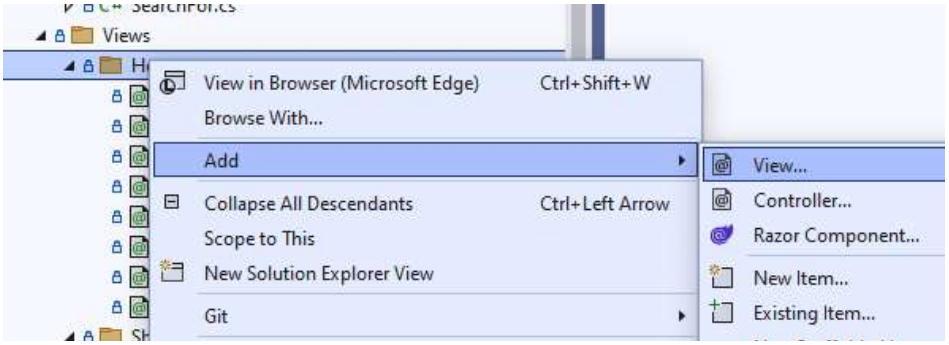


Figure 4 Adding a new View.

3. Razor, ProductViewModel, Details. Name is _ProductCard. It is convention to name a partial view with a leading underscore character.

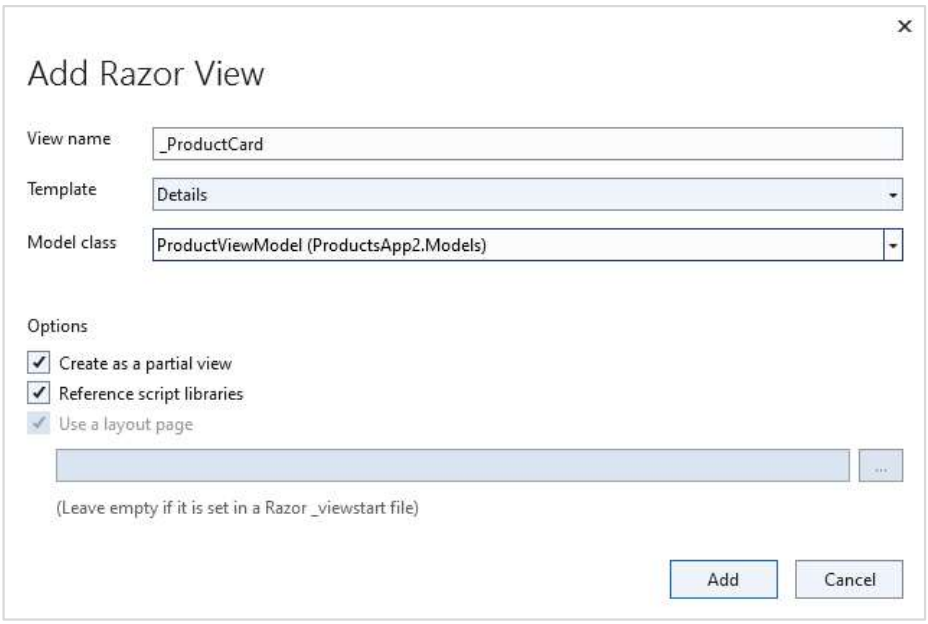


Figure 5 View for "Details" of the ProductViewModel

4. Open ShowAllProductsGrid
5. Add <!—end div - -> at the close of each section. It will avoid confusion when moving blocks of code.

```
@model IEnumerable<ProductsApp2.Models.ProductViewModel>

<p>
    <a asp-action="Create">Create New</a>
</p>

<div class="container">
    <div class="row">
        @foreach (var item in Model)
        {
            <div class="col-md-4">
                <div class="card" style="width: 18rem;">
                    @if (!string.IsNullOrEmpty(item.ImageURL))
                    {
                        <img src="/images/@item.ImageURL" alt="Product Image" onerror="this.onerror=null;this.src='/images/Image_not_
                    }
                    else
                    {
                        <span>No image available</span>
                    }
                    <div class="card-body">

                        <h5 class="card-title">
                            @Html.DisplayFor(modelItem => item.Name)
                            @Html.DisplayNameFor(modelItem => item.Name)
                        </h5>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.Description)
                            @Html.DisplayFor(modelItem => item.Description)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedPrice)
                            @Html.DisplayFor(modelItem => item.FormattedPrice)
                            </p>
                        <p class="card-text">

                            @Html.DisplayNameFor(modelItem => item.FormattedEstimatedTax)
                            @Html.DisplayFor(modelItem => item.FormattedEstimatedTax)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedDateTime)
                            @Html.DisplayFor(modelItem => item.FormattedDateTime)
                            </p>
                        @Html.ActionLink("Edit", "ShowUpdateProductForm", new { id = item.Id }, new { @class = "btn btn-info" })
                        @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-warning" })
                        @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger" })

                    </div> <!-- end card body -->
                </div>  <!-- end Card -->
            </div> <!-- end col-md-4 -->
        }
    </div> <!-- end row -->
</div> <!-- end container -->

```

Figure 6 Added ending tag comments for helping with refactoring.

6. Cut the section from the View.

```
@model IEnumerable<ProductsApp2.Models.ProductViewModel>

<p>
    <a asp-action="Create">Create New</a>
</p>

<div class="container">
    <div class="row">
        @foreach (var item in Model)
        {
            <div class="col-md-4">
                <div class="card" style="width: 18rem;">
                    @if (!string.IsNullOrEmpty(item.ImageURL))
                    {
                        <img src="/images/@item.ImageURL" alt="Product Image" onerror="this.onerror=null;this.src='/images/Image_not_avai
                    }
                    else
                    {
                        <span>No image available</span>
                    }
                    <div class="card-body">

                        <h5 class="card-title">
                            @Html.DisplayFor(modelItem => item.Name)
                            @Html.DisplayNameFor(modelItem => item.Name)
                        </h5>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.Description)
                            @Html.DisplayFor(modelItem => item.Description)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedPrice)
                            @Html.DisplayFor(modelItem => item.FormattedPrice)
                            </p>
                        <p class="card-text">

                            @Html.DisplayNameFor(modelItem => item.FormattedEstimatedTax)
                            @Html.DisplayFor(modelItem => item.FormattedEstimatedTax)
                        </p>
                        <p class="card-text">
                            @Html.DisplayNameFor(modelItem => item.FormattedDateTime)
                            @Html.DisplayFor(modelItem => item.FormattedDateTime)
                            </p>
                        @Html.ActionLink("Edit", "ShowUpdateProductForm", new { id = item.Id }, new { @class = "btn btn-info" })
                        @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-warning" })
                        @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger" })

                    </div> <!-- end card body -->
                </div>  <!-- end Card -->
            </div> <!-- end col-md-4 -->
        }
    </div> <!-- end row -->
</div> <!-- end container-->
```

Figure 7 Selecting the "Card" section of the page.

7. Paste the cut code into the contents of the _ProductCard.cshtml body. Retain the Model. Assign var item = Model in the header of the page or rename all of the instances of "item" on the page.

```
<!-- _ProductCard.cshtml -->

@model ProductsApp2.Models.ProductViewModel

@{
    // either rename all item to Model or use this line to declare item
    var item = Model;
}

<div class="card" style="width: 18rem;">
    @if (!string.IsNullOrEmpty(item.ImageURL))
    {
        <img src="/images/@item.ImageURL" alt="Product Image" onerror="this.onerror=null;this.src='/images/
          Image_not_available.png';" />
    }
    else
    {
        <span>No image available</span>
    }
    <div class="card-body">

        <h5 class="card-title">
            @Html.DisplayFor(modelItem => item.Name)
            @Html.DisplayNameFor(modelItem => item.Name)
        </h5>
        <p class="card-text">
            @Html.DisplayNameFor(modelItem => item.Description)
            @Html.DisplayFor(modelItem => item.Description)
        </p>
        <p class="card-text">
            @Html.DisplayNameFor(modelItem => item.FormattedPrice)
            @Html.DisplayFor(modelItem => item.FormattedPrice)
        </p>
        <p class="card-text">

            @Html.DisplayNameFor(modelItem => item.FormattedEstimatedTax)
            @Html.DisplayFor(modelItem => item.FormattedEstimatedTax)
        </p>
        <p class="card-text">
            @Html.DisplayNameFor(modelItem => item.FormattedDateTime)
            @Html.DisplayFor(modelItem => item.FormattedDateTime)
        </p>
        <a href="#" class="btn btn-info edit-product" data-id="@Model.Id">Edit</a>
        @Html.ActionLink("Details", "ShowProduct", new { id = item.Id }, new { @class = "btn btn-warning" })
        @Html.ActionLink("Delete", "DeleteProduct", new { id = item.Id }, new { @class = "btn btn-danger" })

    </div> <!-- end card body -->
</div>  <!-- end Card -->
```

Figure 8 _ProductCard page now contains the code for rending a Card.

8. Replace the contents of the “card” div in the ShowAllProductsGrid view with the _ProductCard partial view.
9. Add a data-id property to the element. This will put the product id into the HTML.

```
1   <!-- ShowAllProductsGrid.cshtml -->
2   @model IEnumerable<ProductsApp2.Models.ProductViewModel>
3   <p>
4       <a asp-action="Create">Create New</a>
5   </p>
6
7   <div class="container">
8       <div class="row">
9           @foreach (var item in Model)
10          {
11              <div class="col-md-4" data-id="@item.Id">
12                  @Html.Partial("_ProductCard", item)
13              </div> <!-- end col-md-4 -->
14          }
15      </div> <!-- end row -->
16  </div> <!-- end container-->
```

Figure 9 ShowAllProductsGrid is now much shorter. The entire Card section has been replaced with a @Html.Partial command.

10. Run the app. You should see no change in the Grid

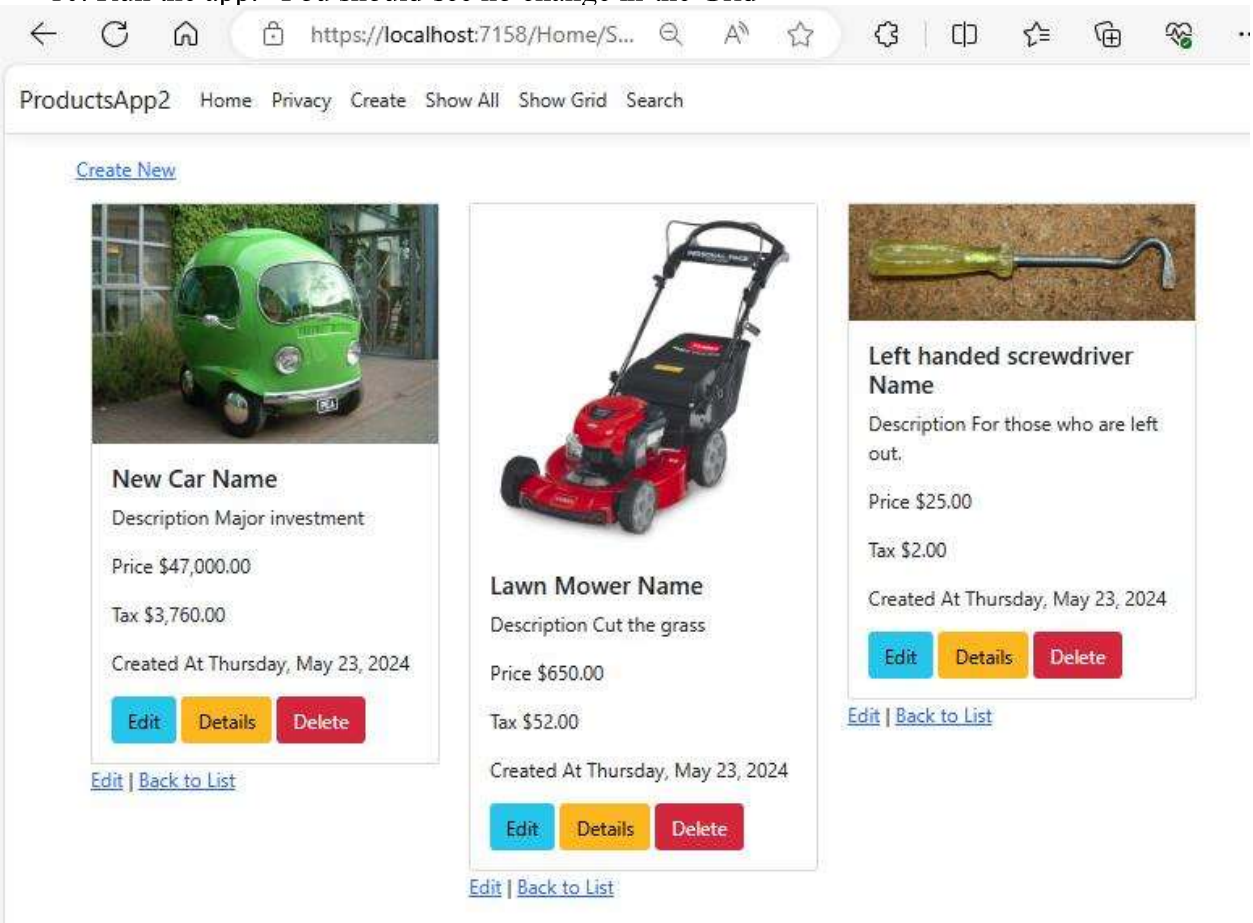


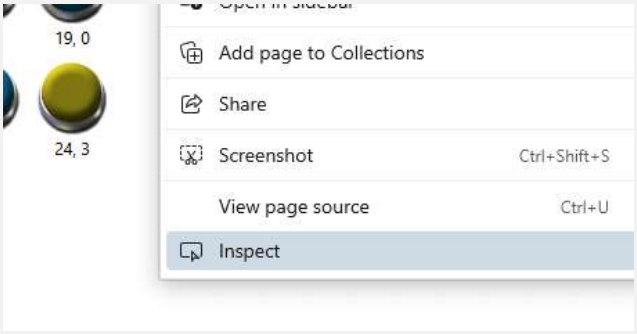
Figure 10 Application continues to run normally. The user does not see any changes.

How to Debug AJAX and JavaScript Issues

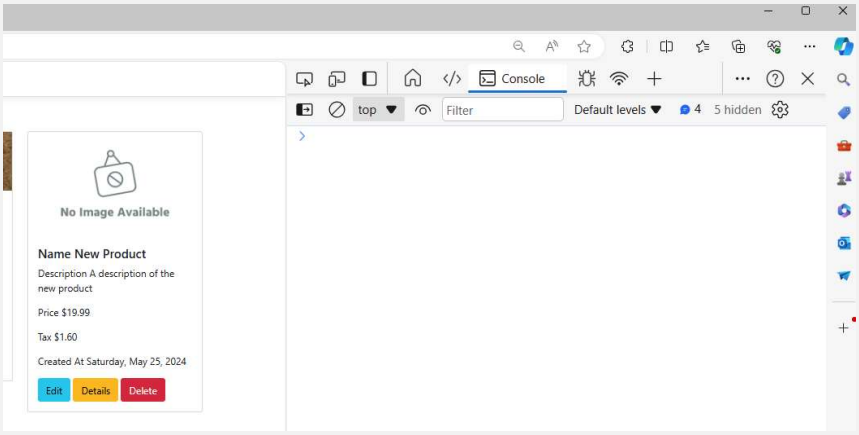
When working with AJAX and JavaScript in ASP.NET Core applications, debugging issues can be challenging but essential to ensure smooth functionality. Here are key strategies and best practices to effectively debug AJAX and JavaScript issues.

1. Use Browser Developer Tools
Modern browsers come equipped with developer tools for debugging JavaScript and AJAX issues. Here's how to use them:

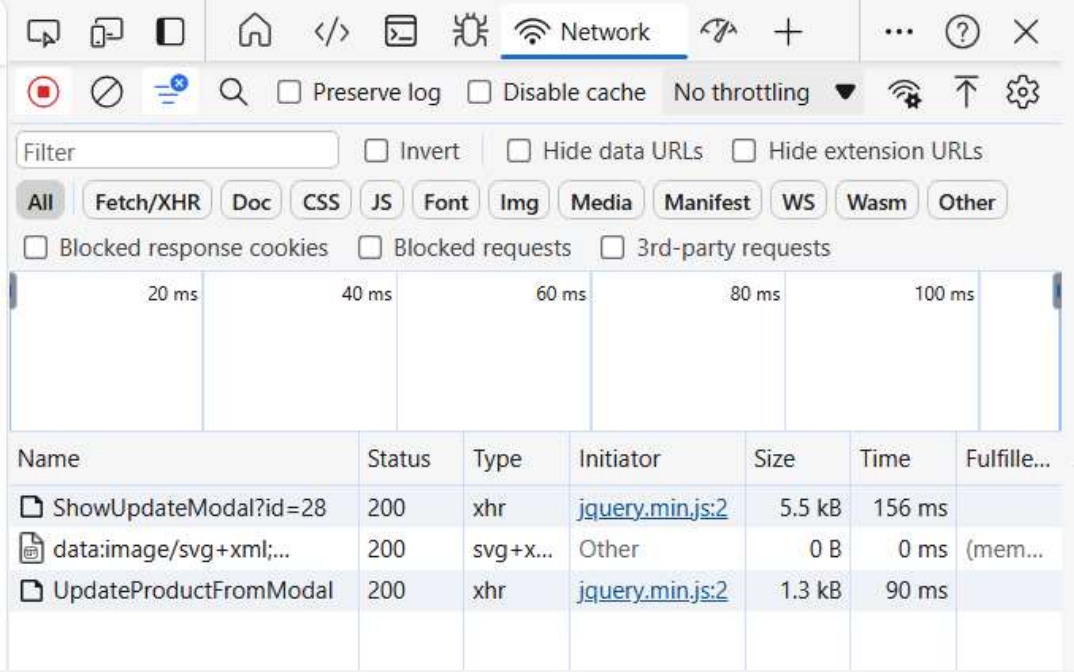
Inspect Elements: Right-click on any element on the page and select "Inspect" to open the developer tools. This allows you to see the HTML and CSS structure.



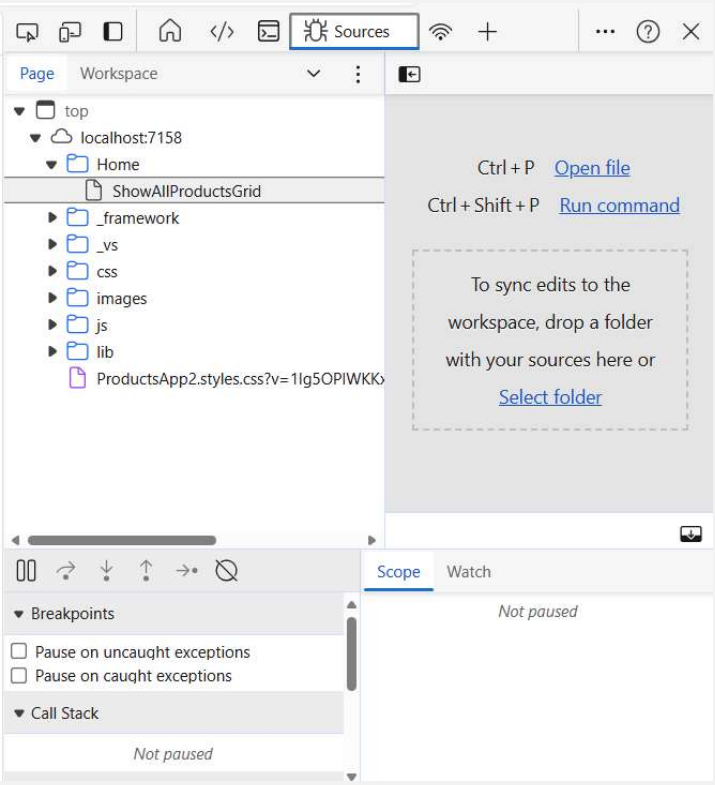
Console: The console is useful for logging messages, errors, and warnings. You can use console.log(), console.error(), and other console methods to print diagnostic messages.



Network Tab: The Network tab shows all network requests made by the page, including AJAX requests. You can inspect the request headers, payload, and response. Look for status codes to identify any failed requests.



Source Tab: The Source tab allows you to view and debug your JavaScript code. You can set breakpoints, step through code, and inspect variables.



2. **Logging and Error Handling in AJAX**
Proper logging and error handling are crucial for identifying and resolving issues with AJAX calls. Here's how to handle errors effectively:

Use the error Callback: When making an AJAX request, you can provide an error callback function to handle any errors that occur. This function can display error messages to the user and log errors for further analysis.

```
$.ajax({
    url: '/Home/ShowUpdateModal?id=' + productId,
    success: function (result) {
        $('#editProductModal .modal-content').html(result);
        var modal = new bootstrap.Modal(document.getElementById('editProductModal'));
        modal.show();
        initializeForm(taxRate);
    },
    error: function (xhr, status, error) {
        console.error('Error: ' + error);
        alert('An error occurred while loading the form: ' + error);
    }
});
```

Server-Side Logging: Ensure that server-side errors are logged appropriately. Use logging frameworks like Serilog, NLog, or the built-in logging in ASP.NET Core to log errors.

```
public IActionResult ShowUpdateModal(int id)
{
    try
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = decimal.Parse( configuration["ProductMapper:TaxRate"]);
        ProductViewModel product =  productService.GetProductById(id).Result;
        return PartialView("ShowUpdateProductForm", product);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while loading the update modal.");
        return StatusCode(500, "Internal server error");
    }
}
```

3. **Debugging AJAX Responses**
Sometimes the issue lies in the data being returned by the server. Use these steps to debug responses:

Check Response Data: In the Network tab of the developer tools, click on the AJAX request and inspect the response data. Ensure it matches the expected format.
Console Logging: Log the response data in your success callback to inspect it.

```
success: function (result) {
    console.log('AJAX Response:', result);
    $('#editProductModal .modal-content').html(result);
    var modal = new bootstrap.Modal(document.getElementById('editProductModal'));
    modal.show();
    initializeForm(taxRate);
}
```

4. **Handling Client-Side Errors**
Ensure that your JavaScript code can handle unexpected scenarios gracefully:
Try-Catch Blocks: While JavaScript does not allow try-catch blocks for AJAX natively, you can use them within your callback functions.

```
try {
    // Code that may throw an error
    someFunction();
} catch (error) {
    console.error('Error:', error);
    alert('An unexpected error occurred: ' + error.message);
}
```

Graceful Degradation: Ensure that your application can still function in a degraded mode if AJAX requests fail. For example, display a user-friendly error message and provide a way to retry the action.

5. **Common Issues and Solutions**
CORS Errors: If your AJAX requests are being blocked due to Cross-Origin Resource Sharing (CORS) policies, ensure that the server is configured to allow requests from your domain.

Network Issues: Slow or unreliable network connections can cause AJAX requests to fail. Implement retries and exponential backoff strategies if necessary.

Syntax Errors: Check for any syntax errors in your JavaScript code that might be preventing it from executing correctly.

Conclusion

Debugging AJAX and JavaScript issues requires a combination of tools and techniques. Use browser developer tools and implement logging and error handling.

Creating a Modal Update Form:

A “modal” form is a popup window that can hide/show dynamically, without requiring a full page refresh.

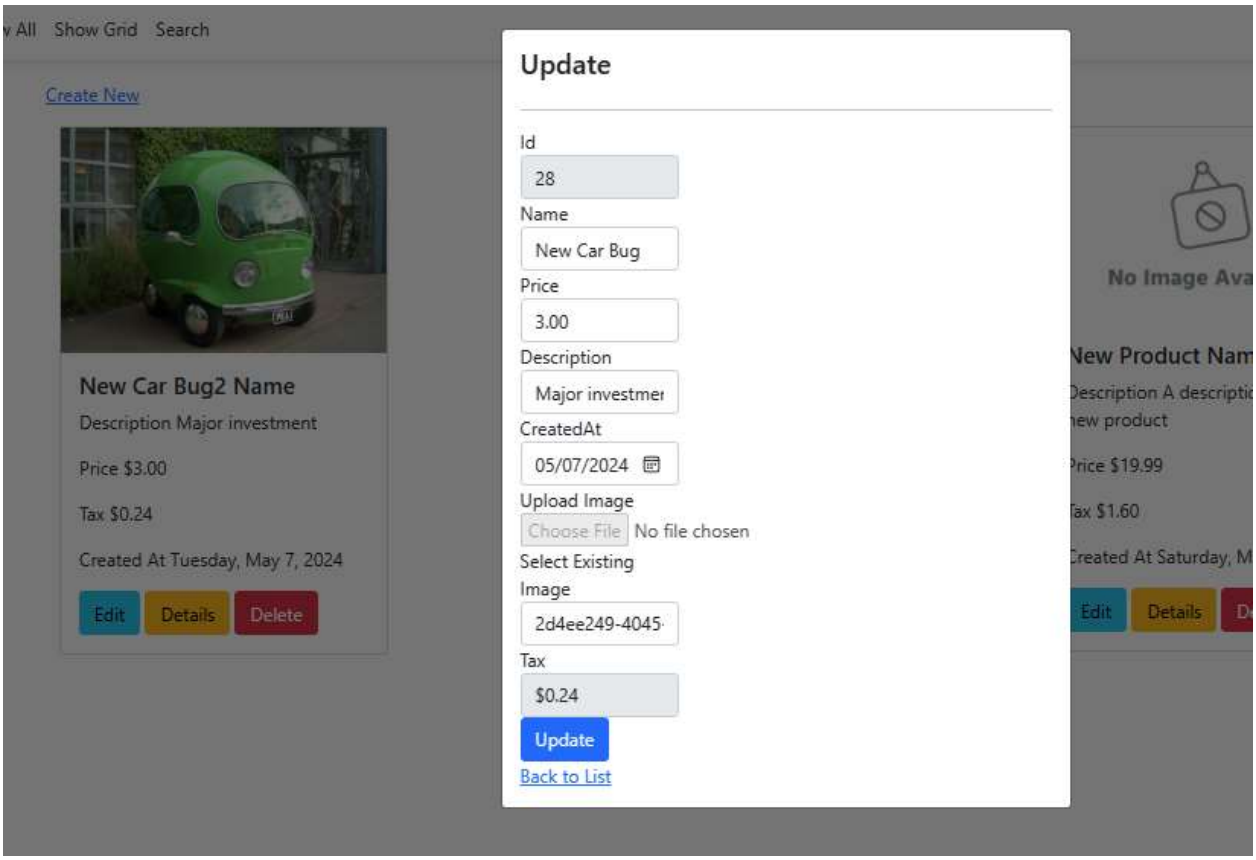


Figure 11 Example of a Modal entry form in action.

1. Update the **Modal** (not “Model”) and JavaScript sections of ShowAllProductsGrid.

```
<!-- Modal for Editing Product . This is hidden until the edit button is clicked. Modal is part of Bootstrap features -->
<div class="modal fade" id="editProductModal" tabindex="-1" role="dialog" aria-labelledby="editProductModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content p-3">
            <!-- Modal content will be loaded here -->
        </div>
    </div>
</div>


@section Scripts {
    <script src="~/js/modalHandler.js"></script>
    <script src="~/js/formHandler.js"></script>
    <script src="~/js/productForm.js"></script>
    <script>
        $(document).ready(function () {
            var taxRate = '@ViewBag.TaxRate';
            bindEditProductEvent(taxRate);
            bindFormSubmission();
        });
    </script>
}
```

Figure 12 Modal and JavaScript sections of ShowAllProductsGrid

2. The purpose of **modalHandler**.js file will be to handle the "**Edit**" button click events from this page.
3. The **formHanlder**.js will handle the "**Submit**" button on Model form.
4. Add a new file modalHandler.js to the js folder.

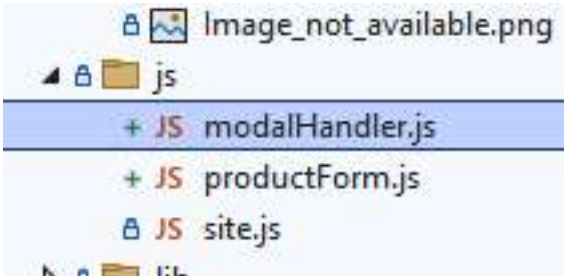


Figure 13 Added a new JavaScript file to the project.

```
// js/modalHandler.js

function bindEditProductEvent(taxRate) {
    // every edit button has a class of edit-product. When clicked, it will trigger the following event
    $(document).on('click', '.edit-product', function (e) {
        e.preventDefault();

        // get the product id from the data-id attribute in the HTML
        var productId = $(this).data('id');
        alert("You clicked the edit button for product id: " + productId + "."); // hide this annoying alert after testing.
        $.ajax({
            // call the ShowUpdateModal action method in the Home controller
            url: '/Home/ShowUpdateModal',
            data: { id: productId },
            success: function (result) {
                alert(result); // hide this annoying alert after testing.

                // result is the HTML content of the modal received from the HomeController

                // replace the modal content with the result
                $('#editProductModal .modal-content').html(result);

                // make sure the image URL is set to the current image
                var currentImage = $('div[data-id="' + productId + '"] img').attr('src');
                currentImage = currentImage.replace("/images/", "").trim();
                $("#ImageURL").val(currentImage);

                // use the Bootstrap modal to show the modal
                var modal = new bootstrap.Modal(document.getElementById('editProductModal'));
                modal.show();

                // Initialize the dynamically loaded form
                initializeForm(taxRate);
            },
            error: function () {
                console.error('An error occurred while loading the form.');
            }
        });
    });
}
```

Figure 14 modalHandler has the job of showing a Modal data entry form using JavaScript.

5. Add an event in the HomeController to correspond to this code.

```
public async Task<IActionResult> ShowUpdateProductForm(int id)
{
    ViewBag.Images = GetImageNames();
    ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
    ProductViewModel product = await _productService.GetProductById(id);
    return View(product);
}
```

Figure 15 ShowUpdateModal method in the HomeController.

6. Run the program. You should be able to show the Modal view of the edit form.

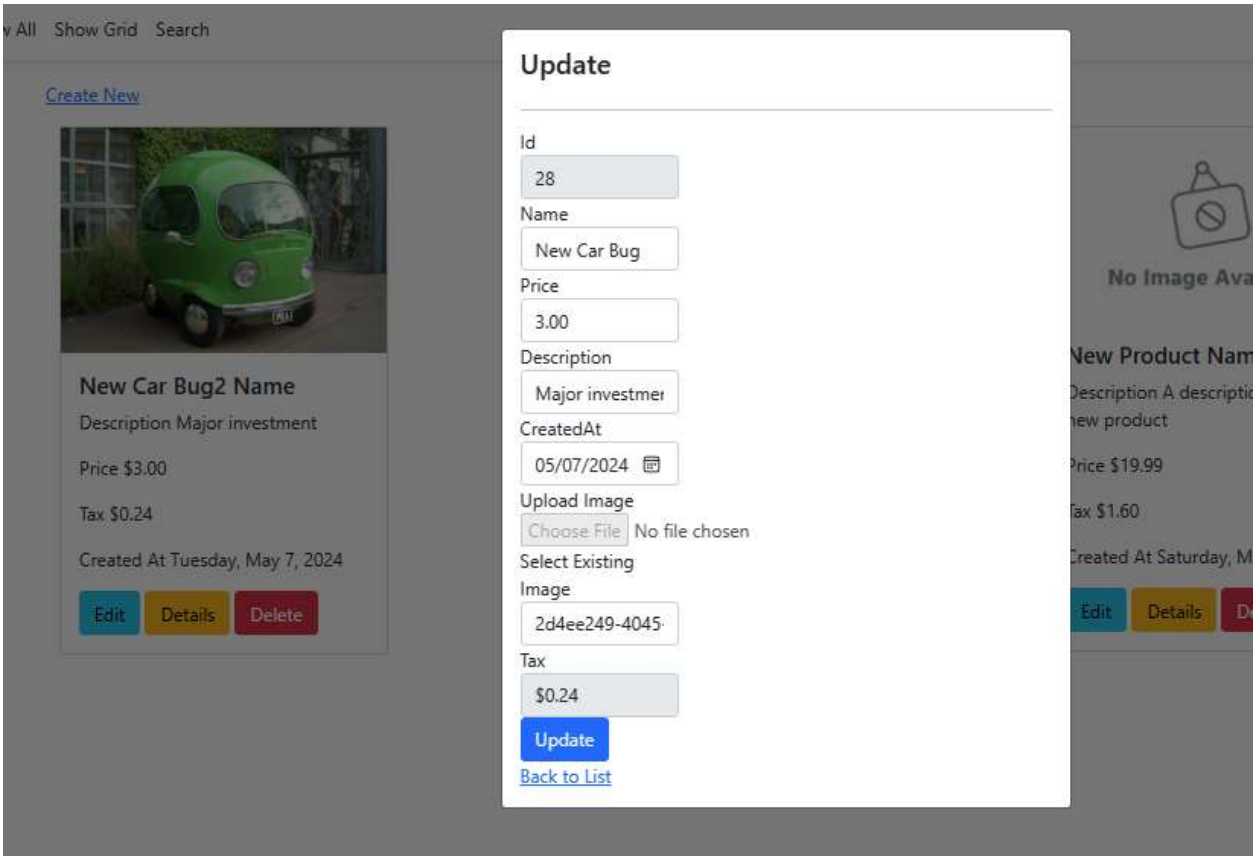


Figure 16 The Edit button now shows a modal instead of a new web page.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

However, when you click the "Update" button, the HomeController processes the UpdateProduct method and the application returns to the index view.

```
[HttpPost]
public async Task<IActionResult> UpdateProduct(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.UpdateProduct(productViewModel);
        return RedirectToAction(nameof(Index));

    }
    else
    // redirect to the create product form if the model state is not valid
    {
        ViewBag.Images = GetImageNames();
        ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
        return View("ShowUpdateProductForm", productViewModel);
    }
}
```

Figure 17 Update Product (unchanged) is the current method that is called when the Update button is clicked. This will not work for our purposes because it redirects the user to the index page, triggering a full page reload.

Understanding AJAX

What is AJAX?
AJAX stands for Asynchronous JavaScript and XML. XML has largely been replaced but the name AJAJ doesn't sound as good, so the original AJAX remains. AJAX is a set of web development techniques that allows web applications to send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page. This means that parts of a web page can be updated without reloading the entire page, resulting in a smoother and more dynamic user experience.

Purpose of AJAX
The primary purpose of AJAX is to improve the interactivity and usability of web applications. By enabling asynchronous data exchange, AJAX allows for:
Faster and More Responsive User Interfaces: Users can interact with the web page without experiencing the delays associated with full page reloads.
Reduced Server Load: By only requesting the data needed, rather than reloading the entire page, AJAX can help reduce the amount of data transferred between the client and server.
Improved User Experience: AJAX allows for real-time updates, such as live search results, form validation, and dynamic content loading, enhancing the overall user experience.

Using AJAX with jQuery
jQuery, a popular JavaScript library, simplifies the use of AJAX in web development. It provides easy-to-use methods for making AJAX requests and handling the responses. Here are some key jQuery AJAX methods:

$.ajax(): The most flexible and powerful method, allowing for detailed configuration of AJAX requests.

$.get(): A shorthand method for making GET requests.
$.post(): A shorthand method for making POST requests.
$.getJSON(): A shorthand method for making GET requests that expect a JSON response.

Birth of AJAX
The term "AJAX" was coined by Jesse James Garrett in 2005, but the underlying technologies had been in use for several years. The key technology enabling AJAX is the XMLHttpRequest object, which was first introduced by Microsoft in Internet Explorer 5 in 1999. This object allowed web pages to make HTTP requests to the server without reloading the entire page.

Evolution of AJAX
2000s: The use of AJAX began to grow rapidly, especially with the rise of Web 2.0 applications. Google Maps and Gmail are early examples of applications that used AJAX to provide a more interactive experience.

2010s: JSON (JavaScript Object Notation) started to replace XML as the preferred data format due to its simplicity and ease of use with JavaScript.

Modern Times: AJAX is now a standard part of web development. Modern frameworks and libraries like React, Angular, and Vue.js build on AJAX principles but offer more abstraction and ease of use.

Future of Dynamic Updates
The future of dynamic updates in web development is promising, with several trends and technologies shaping the landscape:

Single Page Applications (SPAs): SPAs load a single HTML page and dynamically update content as the user interacts with the app. Frameworks like React, Angular, and Vue.js make building SPAs more accessible and efficient.

WebSockets: WebSockets provide a persistent connection between the client and server, allowing for real-time data exchange. This is ideal for applications that require live updates, such as chat applications and live data feeds.

GraphQL: An alternative to REST APIs, GraphQL allows clients to request exactly the data they need, reducing the amount of data transferred and improving performance.

Progressive Web Apps (PWAs): PWAs combine the best of web and mobile apps. They offer offline capabilities, push notifications, and background data synchronization, providing a seamless user experience.

Server-Sent Events (SSE): SSE allows servers to push updates to the client over a single HTTP connection, which is useful for real-time notifications and updates.

Modal Input:

We want the modal to call a slightly different method to update the product. We need a method that will display the changes immediately on the page without a reload.

1. Create a new method in HomeController

```
// receives an update request from the modal form. returns the updated product card as a string
// does not cause a page refresh. this is an AJAX request
[HttpPost]
public async Task<IActionResult> UpdateProductFromModal(ProductViewModel productViewModel)
{
    if (ModelState.IsValid)
    {
        if (productViewModel.ImageFile != null) // file was uploaded
        {
            productViewModel.ImageURL = await PerformFileUpload(productViewModel);
        }

        // add the product to the database
        await _productService.UpdateProduct(productViewModel);

        // get the updated product from the database with all properties set correctly
        ProductViewModel product = await _productService.GetProductById(int.Parse(productViewModel.Id));

        return PartialView("_ProductCard", product);
    }
    else
        {
        ViewBag.Images = GetImageNames();
            ViewBag.TaxRate = decimal.Parse(_configuration["ProductMapper:TaxRate"]);
            return PartialView("ShowUpdateProductForm", productViewModel);
        }

}
```

Figure 18 An update method tailored specifcally to work with the modal input form.

2. Add a new file to the js folder, formHandler.js

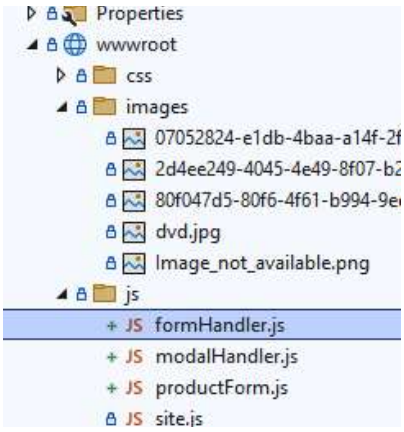


Figure 19 Location of the JavaScript file associated with the formHandling events.

3. Add an event handler to perform AJAX updates for the form submit action.

```javascript
// js/formHandler.js

// This function will bind the form submission event to the form in the modal
function bindFormSubmission() {
    // When a form is submitted, the following event will be triggered
    $(document).on('submit', 'form', function (e) {
        // override the default form submission behavior. We will handle the submission using AJAX
        e.preventDefault();
        alert("Form submit is being handled by AJAX instead of the default form submission."); // hide this annoying alert after testing.
        var form = $(this);
        $.ajax({
            // call the HomeController's UpdateProductFromModal action method
            url: '/Home/UpdateProductFromModal',
            type: 'POST',
            // transform the input fields in the form into a serialized string
            data: form.serialize(),
            success: function (response) {
                alert(response); // hide this annoying alert after testing.

                // response is the HTML content of the updated product card
                var productId = form.find('input[name="Id"]').val();

                // find the product card with the matching data-id attribute and replace it with the updated card
                $('div[data-id="' + productId + '"] .card').replaceWith(response);

                // hide the modal
                var modal = bootstrap.Modal.getInstance(document.getElementById('editProductModal'));
                modal.hide();
            },
            error: function () {
                console.error('An error occurred while updating the product.');
            }
        });
    });
}
```

Figure 20 JavaScript code of formHandler.js

4. Run the program. You should be able to perform in-place updates on a product card. You will notice several things:
    a. A popup alert will show you that the JavaScript code is handling the Edit button click.

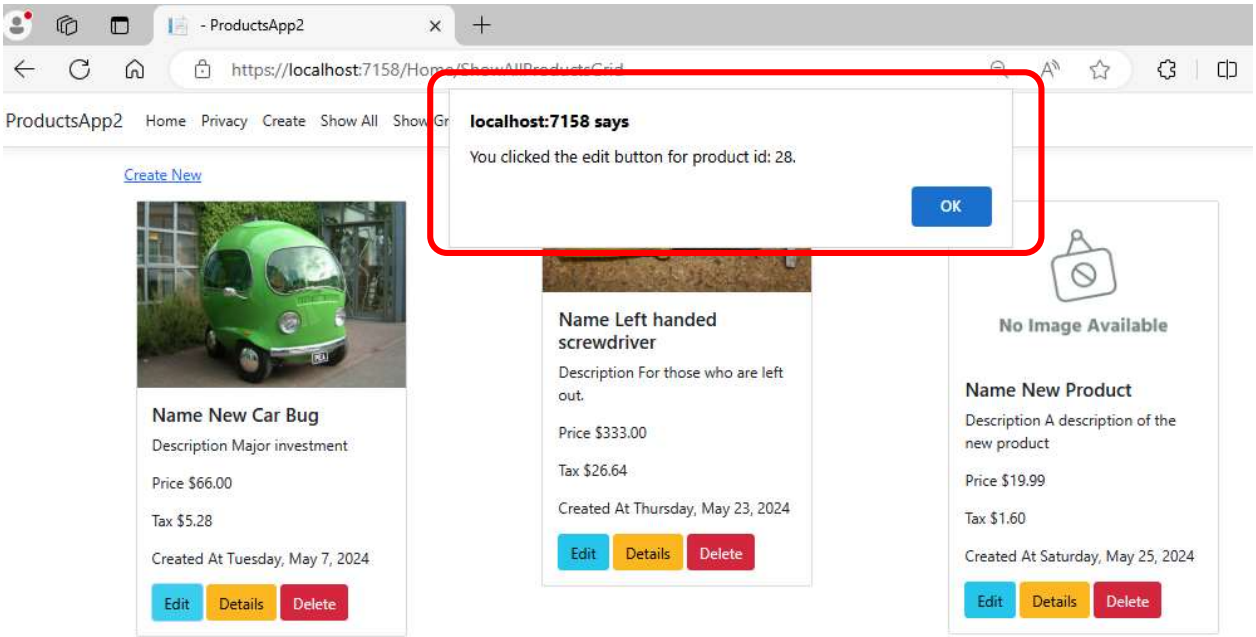


Figure 21 Alert tells the user which item was clicked.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

b. An alert will show the HTML code used to build the Modal dialog.

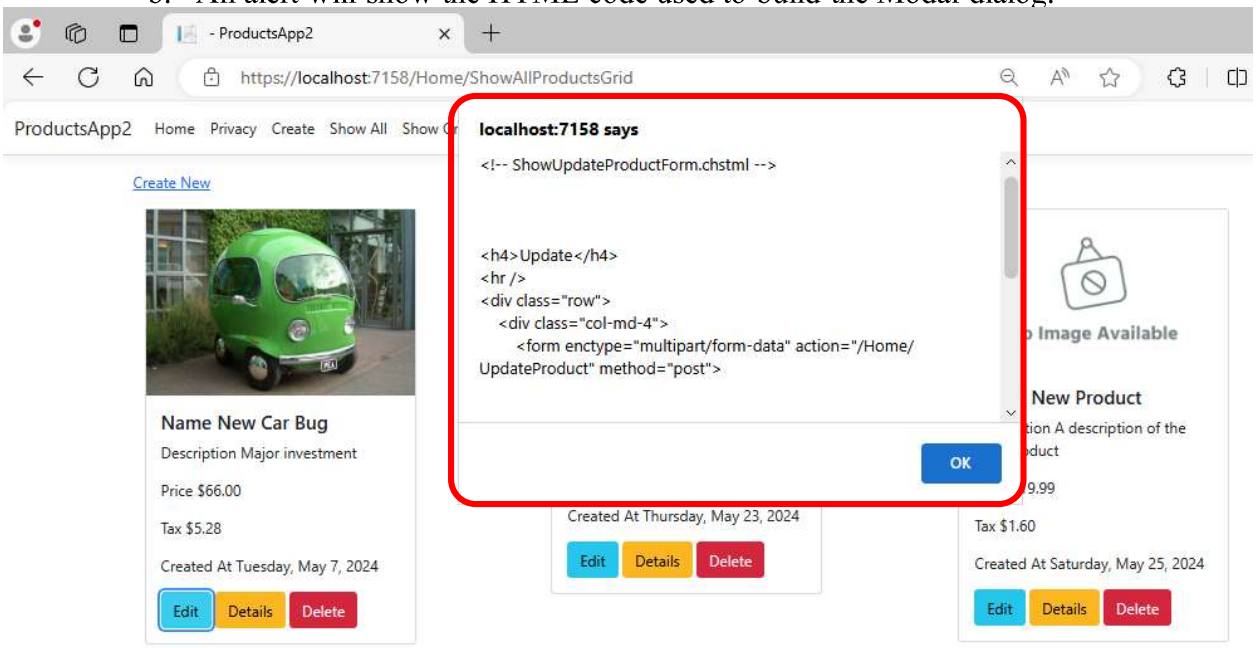


Figure 22 Alert contains the HTML code of the Modal dialog

c. The modal form will function as the previous data entry form.

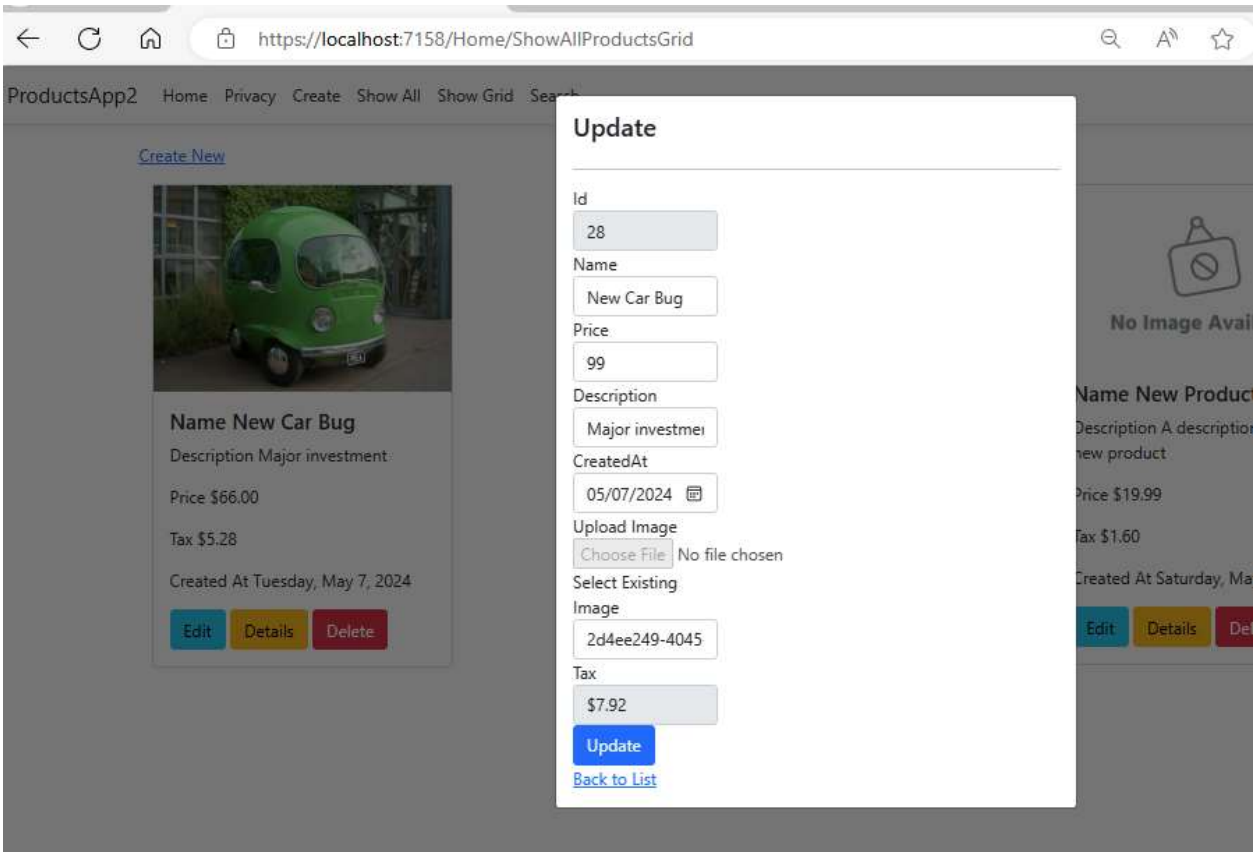


Figure 23 Data entry form for updating a product.

d. Form submit is handled by the JavaScript code.

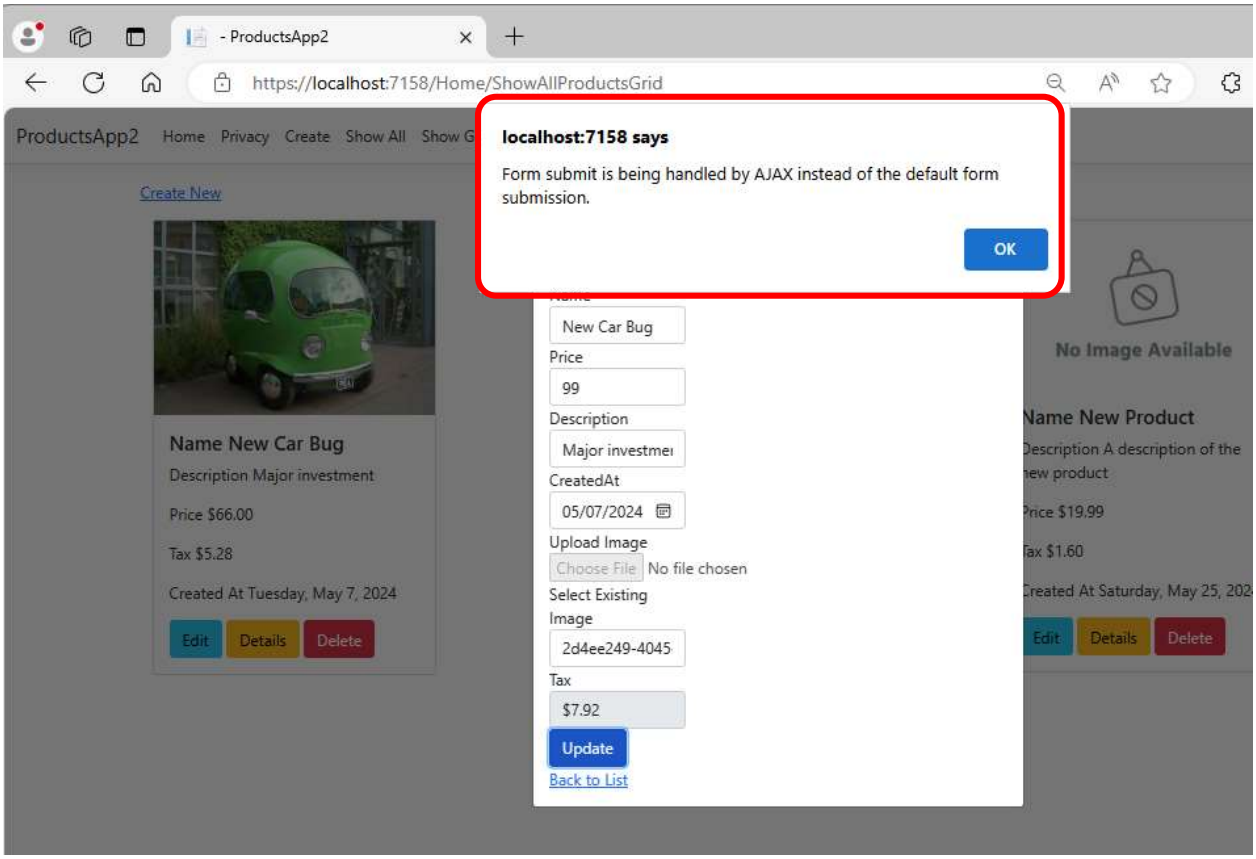


Figure 24 An alert explaining that the Update action is being handled by JavaScript.

e. An alert will show the HTML code used to replace a region of the page.

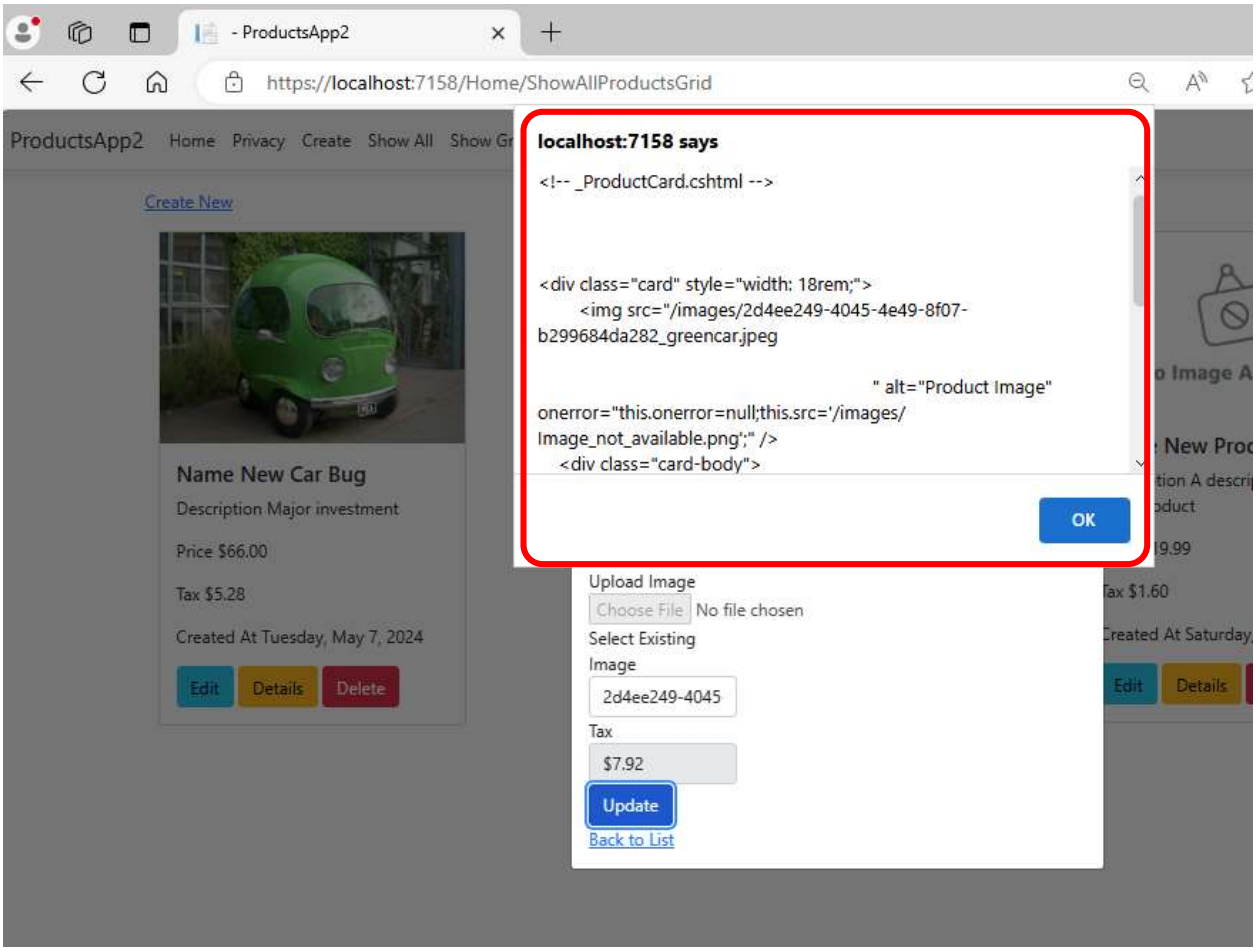


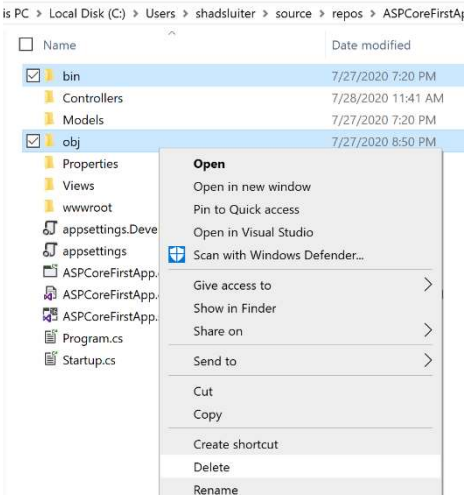
Figure 25 HTML code to replace a card on the products page.

f. The product will show updates without a full-page refresh.

Figure 26 A product will show the updates without a page reload.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Part 2 - Partial Views with the Game Buttons

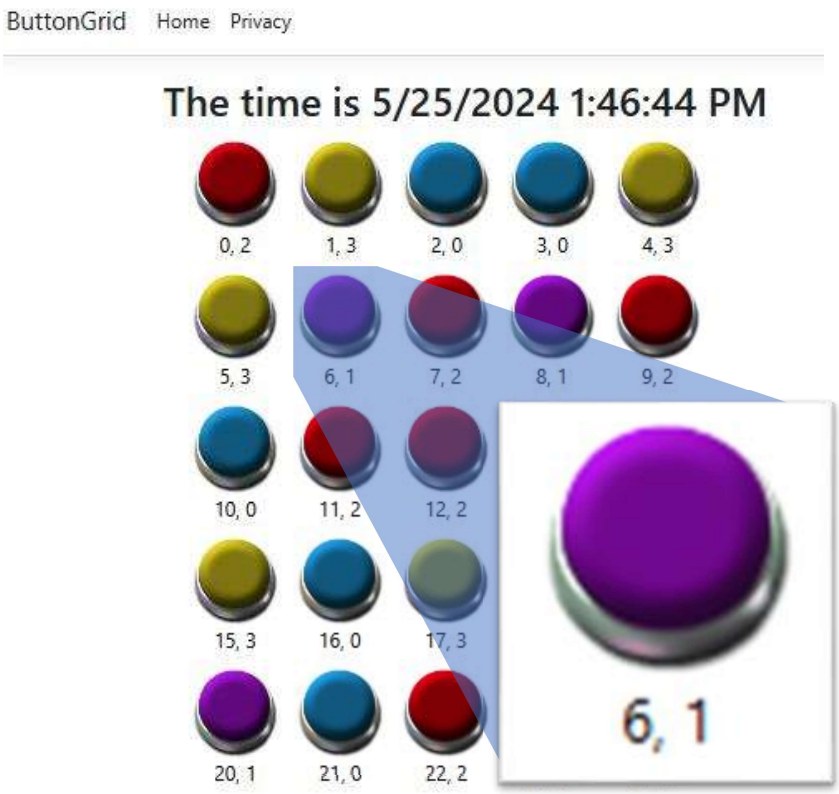


Figure 27 A single button (Partial Page) will be updated without reloading the entire page.

1. Open the Button Grid project.
2. Open the View > Button > Index.cshtml file

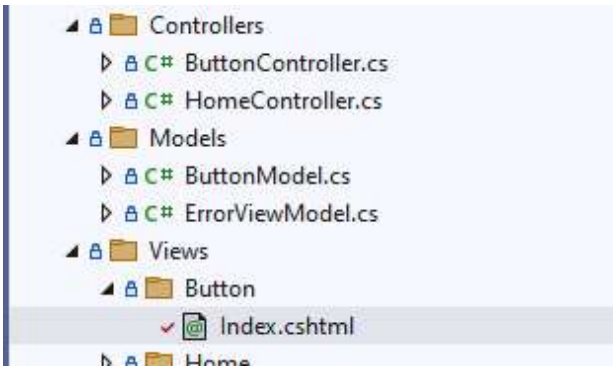


Figure 28 Location of the Index view

```
@model IEnumerable<ButtonGrid.Models.ButtonModel>

<h2>The time is @DateTime.Now</h2>

@for (int i = 0; i < Model.Count(); i++)
{
    var button = Model.ElementAt(i);
    <div class="game-button" data-id="@button.Id">
        <button type="button" name="id" value="@button.Id">
            <img src="/img/@button.ButtonImage" alt="Button" />
            <div>@button.Id, @button.ButtonState</div>
        </button>
    </div>
}
```

"cut" the button code from the index page

Figure 29 Cutting the button Razor code.

3. Replace the button section of the page with a partial view

```
<!-- Index.cshtml -->
@model IEnumerable<ButtonGrid.Models.ButtonModel>

<!-- css from external file button.css -->
<link rel="stylesheet" href="css/buttons.css" />


<h2>The time is @DateTime.Now</h2>
<form asp-action = "ButtonClick">
    <div class="container">
        @for (int i = 0; i < Model.Count(); i++)
        {
            if (i % 5 == 0)
            {
                <div style="width: 100%;"></div>
            }

            var button = Model.ElementAt(i);
            <div class="game-button" data-id="@button.Id">

                @Html.Partial("_Button", button)

            </div>

        }
    </div>
</form>
```

Figure 30 Replacing the button code with a partial view.

4. Create a new file _Button.cshtml in the Views > Button folder

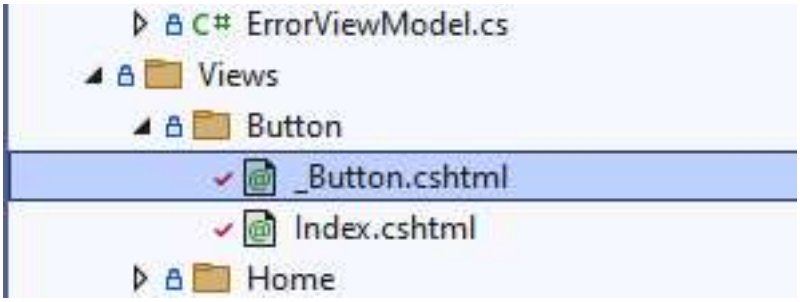


Figure 31 Location of the _Button partial view.

5. Paste the Button code into _Button.cshtml. Rename "button" references with "Model".

```
1  <!-- _button.cshtml -->
2  @model ButtonGrid.Models.ButtonModel
3
4  <button type="button" name="id" value="@Model.Id">
5      <img src="~/img/@Model.ButtonImage" alt="Button" />
6      <div>@Model.Id, @Model.ButtonState</div>
7  </button>
```

Figure 32 Code taken from Index is now in the _Button partial view. Renaming the model variable is necessary.

6. In the ButtonController, add a new method that returns this partial view.

```
32
33          public IActionResult ButtonClick(int id)
34          {
35              ButtonModel button = buttons.FirstOrDefault(b => b.Id == id);
36              if (button != null)
37              {
38                  button.ButtonState = (button.ButtonState + 1) % 4;
39                  button.ButtonImage = butonImages[button.ButtonState];
40              }
41              return RedirectToAction("Index");
42          }
43
44          public IActionResult PartialPageUpdate(int id)
45          {
46              ButtonModel button = buttons.FirstOrDefault(b => b.Id == id);
47              if (button != null)
48              {
49                  button.ButtonState = (button.ButtonState + 1) % 4;
50                  button.ButtonImage = butonImages[button.ButtonState];
51              }
52              return PartialView("_Button", button);
53
54          }
55      }
56  }
```

Figure 33 PartialPageUpdate returns the partial view.

7. In the Index.cshtml file, add the following JavaScript code. This will handle the button click event by retrieving the _Button partial view and swapping out the associated button section.

```
<h2>The time is @DateTime.Now</h2>
<form asp-action = "ButtonClick">
    <div class="container">
        @for (int i = 0; i < Model.Count(); i++)
        {
            if (i % 5 == 0)
            {
                <div style="width: 100%;"></div>
            }

            var button = Model.ElementAt(i);
            <div class="game-button" data-id="@button.Id">

                @Html.Partial("_Button", button)

            </div>

        }
    </div>
</form>

@section Scripts {
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    <script>
        $(document).ready(function () {
            $(document).on("click", ".game-button button", function (e) {
                e.preventDefault();
                var buttonId = $(this).val();
                alert("The button click event is being handled by AJAX instead of the form submit. You clicked item " + buttonId + ".")
                $.ajax({
                    type: "POST",
                    url: "/Button/PartialPageUpdate",
                    data: { id: buttonId },
                    success: function (data) {
                        alert(data);
                        var buttonDiv = $(".game-button[data-id='" + buttonId + "']");
                        buttonDiv.html(data);
                    },
                    error: function (xhr, status, error) {
                        console.error("An error occurred: " + error);
                    }
                });
            });
        });
    </script>
}
```

Figure 34 An AJAX request handles the button clicks, overriding the form submit command.

8. Run the program. You will notice several features of the app if it is running correctly.
   a. The alert messages tell you that the button click event is being handled by the JavaScript code.

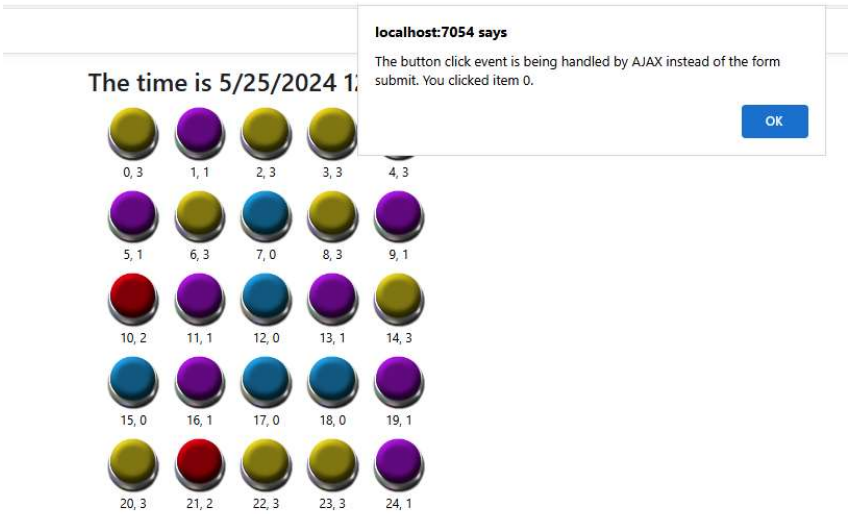


Figure 35 Alert tells us that the JavaScript is handling the button click.

b. The PartialPageUpdate method id the ButtonController is returning just enough HTML code to display a single button.

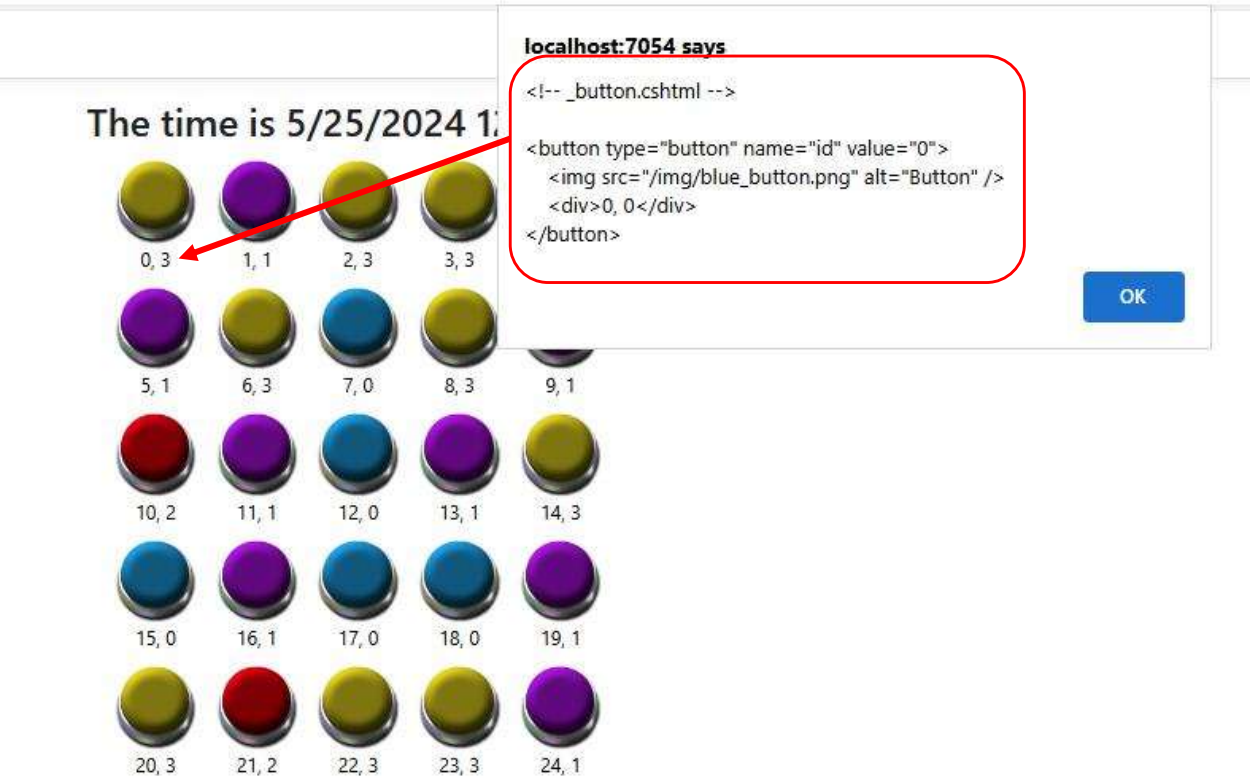


Figure 36 Alert tells us the HTML code that is being sent to the page. It is just enough code for a single button.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

c. A single button is updated but the remainder of the page is not. The time stamp at the page header is created when the application initially launches. However, a button click does NOT cause a full page refresh, leaving the time static.

Figure 37 A single button was updated. The remainder of the page is unchanged.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Conclusions:

What You Learned in this Lesson

Key Points and Their Application

In this lesson, we explored the concept of partial views in ASP.NET Core and how they can be used to modularize and simplify your Razor views. Let's recap the key points:

1. **Understanding Partial Views**:
   - Partial views are reusable Razor views that render a portion of content. They can be embedded within other views, allowing for a more modular and maintainable codebase.
   - **Showcase**: We created a partial view (**_ProductCard.cshtml**) for displaying product details in the ProductApp project.
2. **Creating and Using Partial Views**:
   - We learned how to create partial views and integrate them into our main views using the **Html.Partial** method.

- **Showcase**: In the ProductApp project, we refactored the **ShowAllProductsGrid.cshtml** view to use the **_ProductCard.cshtml** partial view.

3. **Handling AJAX Requests**:
   - AJAX allows parts of a web page to be updated asynchronously without refreshing the entire page, enhancing the user experience.
   - **Showcase**: We implemented AJAX in the ProductApp project to dynamically load and update product details in a modal dialog. The **ShowUpdateModal** action in the HomeController and the corresponding JavaScript code demonstrated this functionality.
4. **Implementing Dynamic Updates**:
   - We handled button click events using AJAX to update parts of the page dynamically, showcasing the power of partial views combined with AJAX.
   - **Showcase**: In the ButtonGrid project, we refactored the **Index.cshtml** view to use the **_Button.cshtml** partial view and implemented AJAX to handle button clicks and update the button state dynamically.

Real-World Applications

Understanding and implementing partial views and AJAX in ASP.NET Core can greatly enhance the user experience and maintainability of your web applications. These skills are highly valuable in real-world job scenarios where modular and responsive web applications are essential. You can apply these skills in various ways:

- **E-commerce Websites**: Use partial views to display product listings, details, and cart updates dynamically.
- **Content Management Systems**: Implement partial views for different sections of a webpage, such as headers, footers, and sidebars, to improve modularity.
- **Interactive Dashboards**: Use AJAX to update data visualizations and reports dynamically without refreshing the entire page.
- **Form Handling**: Create modal forms for CRUD operations, enhancing the user experience by providing a seamless way to create, read, update, and delete records.

Exploring Further Technologies

Now that you have a solid understanding of partial views and AJAX in ASP.NET Core, you may wish to explore other technologies and frameworks that enhance partial page updates and dynamic content loading:

- **React.js or Vue.js**: These JavaScript frameworks provide powerful tools for building dynamic, single-page applications (SPAs) with real-time data updates.
- **Blazor**: An ASP.NET Core framework for building interactive web UIs using C# instead of JavaScript.
- **SignalR**: A library for ASP.NET Core that allows for real-time web functionality, enabling server-side code to push content to clients instantly.

Key Job Search Terms

If you find these skills interesting and want to pursue a career that leverages them, consider looking for job opportunities with the following keywords:

- ASP.NET Core Developer
- Front-end Developer
- Full-stack Developer
- Web Application Developer

- JavaScript Developer
- AJAX Developer

By mastering partial views and AJAX in ASP.NET Core, you have taken a significant step towards becoming proficient in creating modular, maintainable, and responsive web applications. These skills are highly sought after in the tech industry and will open up many opportunities for you in web development.

Check for Understanding:

These questions are not graded but are useful to prepare for upcoming assessments.

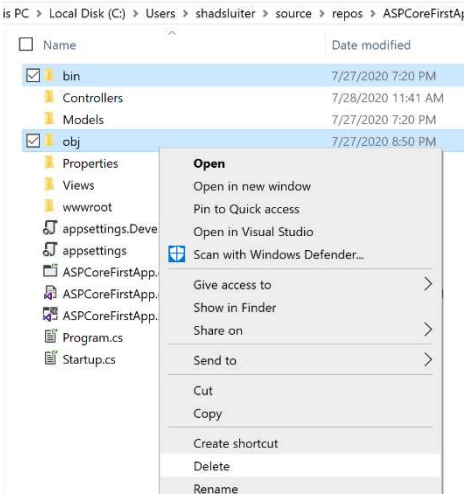
Multiple Choice Questions

1. **What is a partial view in ASP.NET Core?**
   A) A view that displays the entire content of a web page.
   B) A reusable Razor view that renders a portion of content.
   C) A controller that handles HTTP requests.
   D) A JavaScript function used for client-side validation.
2. **Which method is used to render a partial view within a Razor view?**
   A) **Html.RenderPartial**
   B) **Html.Partial**
   C) **Html.RenderAction**
   D) **Html.RenderView**
3. **What are the benefits of using partial views?**
   A) Improved modularity and maintainability
   B) Faster full page reloads
   C) Simplified controller logic
   D) Enhanced database performance
4. **How do you handle a button click event using AJAX in jQuery?**
   A) By using **$.post** to send data to the server.
   B) By using **$.get** to retrieve data from the server.
   C) By using **$.ajax** to send and retrieve data asynchronously.
   D) By using **$.load** to refresh the entire page.
5. **In the ProductApp project, which partial view was created to display product details?**
   A) **_ProductDetails.cshtml**
   B) **_ProductCard.cshtml**
   C) **_ProductInfo.cshtml**
   D) **_ProductView.cshtml**
6. **Which JavaScript library is commonly used for handling AJAX requests in ASP.NET Core applications?**
   A) AngularJS
   B) React.js
   C) jQuery
   D) Vue.js
7. **What is the purpose of the @Html.Partial method in a Razor view?**
   A) To render a view as a string.

B) To execute a controller action.
C) To include the content of a partial view within the main view.
D) To validate form inputs on the client side.

8. **Which of the following is a key feature of AJAX?**
   A) It allows for synchronous web page updates.
   B) It requires a full page reload to update content.
   C) It enables asynchronous updates to parts of a web page.
   D) It only works with static web pages.
9. **In the ButtonGrid project, what does the PartialPageUpdate method in the controller return?**
   A) A JSON object
   B) A full view
   C) A partial view
   D) A string message
10. **Why might you use a modal dialog for editing content in an ASP.NET Core application?**
    A) To display a static message
    B) To refresh the entire page
    C) To provide a dynamic, in-place editing experience without a full page reload
    D) To simplify database queries

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Check for Understanding Answers

1. B
2. B
3. A
4. C

5. B
6. C
7. C
8. C
9. C
10. C