SOFT20181

Internet Application Programming

Supporting Documentation

T0321003

I hereby declare that I am the sole author of this report and that all third-party items including code have been adequately acknowledged and referenced.



Rick Richard Harith

**Table of Contents**

1. Introduction

1.1 Success Criteria

2. Design and Coding Implementation

2.1 UI Design

2.2 User Database

2.2.1 Sign Up

2.2.2 Sign In

2.2.3 Customer Functionalities

2.2.4 Admin Functionalities

2.3 Product Database

2.3.1 Products

2.3.2 Shopping Cart

2.3.3 Search Bar

2.4 Miscellaneous

3. Conclusions

3.1 Challenges Encountered

3.2 Summary and Conclusion

Sources and Reference

Appendix

**1.0 Introduction**

This is the supporting documentation for the e-commerce website created for SOFT20181 Internet Application Programming.

This existing e-commerce website from Term 1 has now been changed to be incorporated into ASP.NET Core. All existing html, css and Javascript have been transferred over, Razor Pages and C# code accompanying it have have been used to reach the requirements of the assessment.

The website has been designed to be mostly media responsive for displays of different sizes on other devices, there are a few errors that will be mentioned. The website is now database driven and features new functionality, this is achieved with Entity Framework Core and SQLlite. User roles that restrict access to certain pages have been added and a shopping cart have also been added.

The website still has a few issues which will be detailed in the report.

**1.1 Success Criteria**

Everything on the website functions as intended with some exceptions which will be discussed in the appropriate sections. There are issues with the design of cart and inefficient coding currently, which will be addressed in the relevant sections below.

|  |  |  |
| --- | --- | --- |
| **Success Criteria Point** | **Priority** | **Implemented?** |
| Database-Driven Content | Required | Yes, users, products and the shopping cart have databases |
| Products with name, description, price and image | Required | **Partially**, products have their names, descriptions, price but not images, the images shown are pre-displayed within the .cshtml, they are not stored within the database, newly added products only have Name, Price and Description. |
| Search Bar | Required | **Yes**, a search bar is implemented and can be used to query specific products based on the product name. |
| Shopping Cart | Required | **Partially, a** shopping cart has been implemented, only customers (logged in users) can add items to the cart but it is not tied to specific customers, when logging out the shopping cart is cleared. |
| Login/Sign Up | Required | **Yes**, the website has a sign up and login page that saves user information into a database |
| User roles | Required | **Yes,** newly signed up users are assigned the appropriate roles and can access certain pages or elements only with said role. |
| Administrator and Admin Pages | Required | **Yes**, An administrator role is present, admins can edit user information as well as add and delete products. |
| Contact Us and About Us Page | Required | **Yes, a** Contact and or About Us page which provides customers with information about the company, what it does, the owners, where it is located, is present |

**2.0** **Design and Coding Implementation**

**2.1 UI Design**

The design of the website is aimed to be minimalistic and straight-forward, with plain colours and rounded fonts, the colours used are light and palatable while being eye catching enough. Elements within the website follow this principle in terms of design, the customisable colours are generally softer colours as well. Because of this, the website was designed to be simple, so it is easy on the users eyes and allows to them to quickly scan and navigate through the website with ease.

* The navigation bar features access to all the pages needed for a user.
* A products page and search page have been implemented for viewing or querying specific products.
* A sign up and sign in page is present, a log out button replaces both if a user is logged in. This means that logged in customers cannot create or sign in again after logging in.
* The home page shows the categories of items available in the website.

Graphical user interface, text

Description automatically generated

Figure . Non-logged In User

Text

Description automatically generated

Figure . Logged In User (Customer)

A picture containing text, appliance

Description automatically generated

Figure . Administrator User

* Different types of users can access different functionalities within the website. For example, non-customers can only view products, but customers can add the products to a shopping cart.
* A shopping cart is available to logged in customers, customers can see the products added and the quantity. The cart can be edited and logging out clears it.
* An administrator account exists and the log in information is as follows:  
  Username: Admin  
  Password: @Admin12345

The account can access pages to edit customer information and delete or add products.

Passwords of customers are not shown and cannot be edited for privacy reasons.

Overall, the website was designed to be easily navigated so users will not be confused by the layout, it conforms to usual e-commerce website practises of having a navigation bar, search bar and products page with appropriate extra pages to give information on the website and contact information. Most e-commerce websites are typically very cluttered, the website is designed to not have too many sub menus to clutter up the design, but arguably, it requires more user input to navigate.

**2.2 User Database**

namespace Coursework.Pages.Model

{

public class UserData : IdentityUser

{

[Required(ErrorMessage = "Name is required.")]

public string Name { get; set; }

[Required(ErrorMessage = "Email is required.")]

[EmailAddress(ErrorMessage = "Invalid email format.")]

public override string Email { get; set; }

[Required(ErrorMessage = "Password is required.")]

public string Password { get; set; }

}

}

UserData.cs is a model class file that inherits from ‘IdentityUser’ class that is provided by Microsoft.AspNetCore.Identity, it has three properties called ‘Name’, ‘Email’ and Password.

The Identity class inherited enables user authentication and authorization and the three properties represent the data that will be stored in the database. Inheriting from the identity class also automatically generates a ‘UserName’ and ‘Id’ property.

public class UserDataContext : IdentityDbContext<UserData>

{

public UserDataContext(DbContextOptions<UserDataContext> options) : base(options) { }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<UserData>().HasIndex(u => u.Email).IsUnique();

}

public DbSet<UserData> userData { get; set; }

}

UserDataContext.cs is the database context for the ‘UserData’ class, it also extends the ‘IdentityDbCOntext’ class. The IdentityDbContext<UserData> generic parameter specifies that the UserData class represents user data in the context of identity authentication and authorization.

The ‘OnModelCreating’ method is overridden so when the database is built, e-mail is configured to be UNIQUE. This ensures users won’t have duplicate e-mails or accounts with the same email. Finally, ‘DbSet’ enables CRUD operations in the table, so the entities within <UserData> can be manipulated.

In the Program.cs contains this code:

var Configuration = builder.Configuration;

builder.Services.AddDbContext<UserDataContext>(options =>options.UseSqlite(builder.Configuration.GetConnectionString("Default")));

builder.Services.AddIdentity<UserData, IdentityRole>().AddEntityFrameworkStores<UserDataContext>();

This code configures the dependency injection and sets up the database, the authentication system is also set up for the database.

using (var scope = app.Services.CreateScope())

{

var roleManager = scope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();

await roleManager.CreateAsync(new IdentityRole("Customer"));

await roleManager.CreateAsync(new IdentityRole("Admin"));

}

This block of code creates two new roles called ‘Customer’ and ‘Admin’ in the Identity system, it uses the ‘RoleManager’ service to do this. Adding this code to Program.cs initialises the Identity system with the roles needed for the website.

Finally, this code lies within the ‘appsettings.json’:

"ConnectionStrings": {

"Default": "Data Source=userData.db",

This code specifies the connection string for the SQLlite database. Specifically, it uses the userData.db and the connection is called ‘Default’, the Program.cs code from earlier gets this connection string.

All this code is needed to build the ‘userData.db’ database, by running these commands in the Package Manager Console:

add-migration userData -context UserDataContext  
update-database -context UserDataContext

**2.2.1 Sign Up**

This section will detail how the sign up page requirements were achieved step by step. Below is the code for the sign up page’s C# code behind file, ‘SignUp.cshtml.cs’:

private readonly UserDataContext \_userDataContext;

private readonly UserManager<UserData> \_userManager;

public SignUpModel(UserDataContext userDataContext, UserManager<UserData> userManager, SignInManager<UserData> signInManager)

{

\_userDataContext = userDataContext;

\_userManager = userManager;

users = new UserData();

}

This is a constructor for the class which takes three parameters which are ‘UserDataContext’, ‘UserManager<UserData>’ and ‘SignInManager<UserData>’. DbContext is used to manipulate data within the ‘userData.db’ database, UserManager is used to manage user accounts. The ‘users’ object is initialised for the sake of a new instance of the UserData(); class to ensure that it is used as a property in ‘BindProperty’ attribute.  
  
[BindProperty]

public UserData users { get; set; }

If the ‘users’ object is not initialised, then it will return null and cause an error within the ‘OnPostAsync’ method. Speaking of which, is below:  
  
 public async Task<IActionResult> OnPostAsync()

{

if (!ModelState.IsValid)

{

return Page();

}

try

{

var Result = await \_userManager.CreateAsync(users, users.Password);

if (Result.Succeeded)

{

await \_userManager.AddToRoleAsync(users, "Customer");

TempData["SignUpSuccessMessage"] = "Sign up successful! You can now log in!";

return RedirectToPage("/Login");

}

else

{

foreach (var error in Result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

return Page();

}

}

The ‘OnPostAsync’ method uses Async instead of the normal ‘OnPost’ because processing requests does not block the main thread, it is encouraged to be used when calling databases or external APIs. It also improved performance of the application because it utilises resources such as the CPU better.

Besides that, the ‘OnPostAsync’ method is called when a user submits the form in the page. When that happens, it checks if the ‘ModelState’ is valid, if it is not, then reload the page. The ‘ModelState’ contains validation for objects submitted by forms, so it can detect any errors that may occur during validation. If the ‘ModelState’ is valid, then the code creates a new user using the ‘CreateAsync’ method from ‘UserManager’ based on the information put into the ‘users’ object. If the user is successfully created, then the role ‘Customer’ is added to said user. A success message is stored within ‘[TempData]’ and the user is redirected to the Log In page. The ‘else’ statement uses a for loop to print all errors that happen from ‘ModelState’ and reloads the page.

The code is wrapped in ‘try’ and ‘catch’ because of the possibility of the UNIQUE key of E-mail being violated. So this code catches that error and reloads the page with appropriate error messages:

catch (DbUpdateException ex)

{

var sqlException = ex.InnerException as SqliteException;

if (sqlException?.SqliteErrorCode == 19)

{

ModelState.AddModelError(string.Empty, "The email address is already in use.");

}

else

{

ModelState.AddModelError(string.Empty, "An error occurred while processing your request. Please try again later.");

}

return Page();

}

To link the code to the ‘OnPostAsync’ method, the form has <method = “post”>, the information will be appended to the ‘users’ object with <asp-for= = “@Model.users.-“> and errors are handled by <asp-validation-summary=”All”>

<**form** method="post">

<**div** **asp-validation-summary**="All" class="text-failure"></**div**>

<**asp-for**="@Model.users.UserName" required>

**2.2.2 Sign In**

Once the sign up is successful, users are directed to the Login page with a message displayed from the [TempData] mentioned earlier. Here is an explanation of the code behind file ‘Login.cshtml.cs’

[TempData]

public string SignUpSuccessMessage { get; set; }

private readonly CartDataContext \_cartDataContext;

private readonly SignInManager<UserData> \_signInManager;

public LoginModel(SignInManager<UserData> signInManager, CartDataContext cartDataContext)

{

\_signInManager = signInManager;

\_cartDataContext = cartDataContext;

}

public class LoginViewModel

{

[Required]

public string Username { get; set; }

[Required]

public string Password { get; set; }

}

[BindProperty]

public LoginViewModel LoginInput { get; set; }

This code has the same logic to the sign up page, but it also has a new ‘cartDataContext’ and ‘SignInManager’ which is from the Identity authentication system, this will be used to clear the table that saves the customers chosen products with when they press the ‘Log Out button’, which will be explained later.

public async Task<IActionResult> OnPostAsync(string returnUrl = null)

{

if (ModelState.IsValid)

{

var result = await \_signInManager.PasswordSignInAsync(LoginInput.Username, LoginInput.Password, isPersistent: false, lockoutOnFailure: false);

if (result.Succeeded)

{

TempData["LoginSuccessMessage"] = "Login successful! Start Shopping Now!";

return RedirectToPage("/Index");

}

ModelState.AddModelError(string.Empty, "Invalid login attempt.");

}

return Page();

}

The ‘OnPostsync’ method for the login code uses the same ‘ModelState’ to validate inputs, the difference is the use of the ‘SignInManager’ that signs the user in based on ‘LoginInput.Username’ (Username) and ‘LoginInput.Password’ (Password). ‘IsPersistent’ is a Boolean that determines if the authentication session should persist across multiple requests. ‘lockoutOnFailure’ determines if the user should be locked out if failed login attempts are failed too much.

Finally, an ‘OnPostLogoutAsync’ method was created to handle user log outs, the button only appears if users are authenticated and when pressed, executes the following code to log the user out:

public async Task<IActionResult> OnPostLogoutAsync()

{

await \_signInManager.SignOutAsync();

\_cartDataContext.ShoppingCartItems.RemoveRange(\_cartDataContext.ShoppingCartItems);

await \_cartDataContext.SaveChangesAsync();

TempData["LoginSuccessMessage"] = "You're now logged out, we'll miss you...";

return RedirectToPage("/Index");

}

It also clears the cart with ‘.RemoveRange’ to ensure that the next user logged in will have an empty cart. A message saved in [TempData] is again used to print out in the Home page so the user is aware of the log out. This log out button appears in the ‘\_Layout.cshtml’ navigation bar with this code to link it to the ‘OnPostLogOutAsync’ method:

<li class="navbar-item">

<**form** method="post" **asp-page-handler**="Logout" **asp-page**="/Login" onsubmit="return confirm('Are you sure you want to log out? Your cart will be cleared!');">

<button type="submit" class="nav-logout">Log Out</button>

</**form**>

</li>

**2.2.3 Customer Functionalities**

This section will detail what signed in Customers can do within the website and the accompanying code that helps explain it.

Firstly, non-logged in users will not be able to see the ‘Add To Cart’ button on products. This is done with C# if statements throughout the code. For example in the ‘Products.cshtml’:

@if (User.Identity.IsAuthenticated && User.IsInRole("Customer"))

{

<**form** method="post">

<input type="hidden" name="productId" value="@product.ProductId" />

<button type="submit" class="add-to-cart-2">Add to Cart</button>

</**form**>

}

This button is hidden if the user does not meet the if statement requirements. Non-logged in users are also unable to access the ‘ShoppingCart.cshtml’ as the button does not appear in the navigation bar as seen below:

@if (User.Identity.IsAuthenticated && User.IsInRole("Customer"))

{

<li class="navbar-item">

<**a** **asp-page**="/ShoppingCart" **asp-area**="" class="nav-link">Shopping Cart</**a**>

</li>

<li class="navbar-item">

<**form** method="post" **asp-page-handler**="Logout" **asp-page**="/Login" onsubmit="return confirm('Are you sure you want to log out? Your cart will be cleared!');">

<button type="submit" class="nav-logout">Log Out</button>

</**form**>

</li>

}

The code above is from ‘\_Layout.cshtml’, which is responsible for the consistent appearance of <headers> and <footers> across all pages as well as handling <head> code. Even if users try to enter restricted pages by typing so in the address bar, the authorisation system ensures that users cannot access said pages because of this code in each page:  
 [Authorize(Roles="Customer")]

In conclusion, logged in customers are able to add products to a shopping cart and view the hidden add to cart button that appears only when logged in.

**2.2.4 Admin Functionalities**

Two extra pages exist for Administrators of the site, these two pages are restricted with the [Authorisation] attribute limited to the admin role. By navigating to the Log In page, the admin account can be accessed with:

Username: Admin  
Password: @Admin12345

‘AddProducts.cshtml’ and ‘Admin.cshtml’ are responsible for adding or deleting products and editing user information or deleting users. This section of the code will detail the code behind these pages, starting with ‘Admin.cshtml’  
  
private readonly UserManager<UserData> \_userManager;

public AdminModel(UserManager<UserData> userManager)

{

\_userManager = userManager;

}

public IList<UserData> Users { get; set; }

public async Task OnGetAsync()

{

Users = await \_userManager.GetUsersInRoleAsync("Customer");

}

This code has the same logic of injecting an instance ‘UserManager<UserData>’ into a private field, the ‘OnGetAsync’ method retrieves all users with the customer role and puts it into the list named ‘Users’ to be displayed in the cshtml.  
<td>

<**form** method="post">

<input type="hidden" name="id" value="@user.Id" />

<input type="text" name="username" value="@user.UserName" />

</**form**>

</td>

<td>

<**form** method="post">

<input type="hidden" name="id" value="@user.Id" />

<input type="text" name="name" value="@user.Name" />

</**form**>

</td>

<td>

<**form** method="post">

<input type="hidden" name="id" value="@user.Id" />

<input type="email" name="email" value="@user.Email" />

</**form**>

</td>

<td>

<**form** method="post" **asp-page-handler**="Delete" onsubmit="return confirm('Are you sure you want to delete this user?');">

<input type="hidden" name="id" value="@user.Id" />

<button type="submit" class="delete-button" value="Delete">Delete</button>

</**form**>

</td>  
  
In the cshml is this code which displays each of the users information as text inputs to be edited, this is so each field can be changed and trigger the ‘OnPostAsync’ method within the code behind file.

public async Task<IActionResult> OnPostAsync(string id, string username, string name, string email, string password)

{

var user = await \_userManager.FindByIdAsync(id);

if (user == null)

{

return NotFound();

}  
The method takes in four parameters that is present in <UserData> and uses UserManager to find each user by Id. For each input field, this code is used to update user info when the form is submitted

if (!string.IsNullOrEmpty(username))

{

user.UserName = username;

}

if (!string.IsNullOrEmpty(name))

{

user.Name = name;

}

if (!string.IsNullOrEmpty(email))

{

user.Email = email;

}

var result = await \_userManager.UpdateAsync(user);

if (result.Succeeded)

{

TempData["LoginSuccessMessage"] = "User updated!";

return RedirectToPage();

}

‘.UpdateAsync’ is used to update the information within the database. To make sure the administrator does not violate the UNIQUE key within a database for Username and E-mail, this code was written to restore the previous entries if changed

bool emailChanged = !string.IsNullOrEmpty(email) && email != user.Email;

bool usernameChanged = !string.IsNullOrEmpty(username) && username != user.UserName;

// Check if email or username already exists in the database

if (emailChanged)

{

var existingUserWithEmail = await \_userManager.FindByEmailAsync(email);

if (existingUserWithEmail != null && existingUserWithEmail.Id != user.Id)

{

email = user.Email; // Restore the original email in the form

}

}

if (usernameChanged)

{

var existingUserWithUsername = await \_userManager.FindByNameAsync(username);

if (existingUserWithUsername != null && existingUserWithUsername.Id != user.Id)

{

username = user.UserName; // Restore the original username in the form

}

}

An ‘OnPostDeleteAsync’ method has also been written to delete customer users  
public async Task<IActionResult> OnPostDeleteAsync(string id)

{

var user = await \_userManager.FindByIdAsync(id);

if (user == null)

{

return NotFound();

}

var result = await \_userManager.DeleteAsync(user);

if (result.Succeeded)

{

TempData["LoginSuccessMessage"] = "User deleted!";

return RedirectToPage();

}

else

{

foreach (var error in result.Errors)

{

ModelState.AddModelError("", error.Description);

}

return Page();

}

}

The user is deleted with the UserManager ‘.DeleteAsync’ command.

The ‘AddProducts.cshtml’ functions the same, but products can only be added and deleted, editing is not possible. Here is the code responsible for adding products into the Products daabase:

await \_productContext.Products.AddAsync(Product);

await \_productContext.SaveChangesAsync();

Below the Add Product form is a display of the list of products available, the original 13 products that were present from the first assessment cannot be deleted as unique cshtml pages are linked to said pages. The only products that can be deleted are newly added products. The deleting code is the same as the user delete code but for products instead.

Only customers are able to add items to cart so the administrator does not have access to the shopping cart.

**2.3 Product Database**

Same as before, the product database is created with ‘Products.cs’ and ‘ProductDataContext.cs’  
  
namespace Coursework.Pages.Model

{

public class Products

{

[Key]

public int ProductId { get; set; }

[Required(ErrorMessage = "Product name is required.")]

public string Name { get; set; }

[Required(ErrorMessage = "Product description is required.")]

public string Description { get; set; }

[Required(ErrorMessage = "Product price is required.")]

public decimal Price { get; set; }

}

}

The attribute ‘[Key]’ is set to ProductId so it functions as the primary key, the ‘[Required]’ attribute is used to set an error message when adding products for the admin.   
  
using Microsoft.EntityFrameworkCore;

namespace Coursework.Pages.Model

{

public class ProductDataContext : DbContext

{

public ProductDataContext(DbContextOptions<ProductDataContext> options) : base(options) { }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Products>().HasIndex(u => u.Name).IsUnique();

}

public DbSet<Products> Products { get; set; }

}

}

Products inherits from DbContext instead, and the ‘OnModelCreating’ method is overridden to set the ‘Name’ property as unique so no duplicate products exist when adding products. The set up for the product database is the same as before in the ‘Program.cs’ and ‘appsettings.json’

builder.Services.AddDbContext<ProductDataContext>(options => options.UseSqlite(Configuration.GetConnectionString("Products")));

"ConnectionStrings": {

"Default": "Data Source=userData.db",

"Products": "Data Source=products.db",  
The database is then generated with Package Manager Console.

**2.3.1 Products**

The code for getting a list of products from the database is similar to listing the users in the ‘userData.db’  
  
private readonly ProductDataContext \_productContext;

private readonly CartDataContext \_cartContext;

public IList<Products> ProductList { get; set; }

public ProductsModel(ProductDataContext productContext, CartDataContext cartContext)

{

\_productContext = productContext;

\_cartContext = cartContext;

}

public async Task<IActionResult> OnGetAsync()

{

ProductList = await \_productContext.Products.ToListAsync();

return Page();

}

The ‘OnGetAsync’ method is called and using the ‘.ToListAsync’ method available in the DbContext system, the products can be fetched to be displayed in the cshtml. Using a for loop, the products can be displayed from the database in the cshtml like so:

@foreach (var product in Model.ProductList)

{

<div class="product-container-2">  
<div class="product-name-2">@product.Name</div>

<div class="product-description-2">@product.Description</div>

<div class="product-price-2">@product.Price.ToString("$0.00")</div>

}

Then the products are styled with css to display neatly in the products page.

**2.3.2 Shopping Cart**

To explain the shopping cart, a new database is created specifically for storing user added products. Creating the ‘Cart.db’ is exactly the same as the products database.

CartData.cs:  
namespace Coursework.Pages.Model

{

public class Cart

{

[Key]

public int ProductId { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int Quantity { get; set; }

}

}

CartDataContext.cs:  
namespace Coursework.Pages.Model

{

public class CartDataContext : DbContext

{

public CartDataContext(DbContextOptions<CartDataContext> options) : base(options) { }

public DbSet<Cart> ShoppingCartItems { get; set; }

}

}

Once the database is created, the code below details how exactly the ‘Add To Cart’ button functions:

public async Task<IActionResult> OnPostAsync(int productId)

{

// Get the product from the product context

var product = await \_productContext.Products.FindAsync(productId);

// Get the shopping cart item from the cart context

var cartItem = await \_cartContext.ShoppingCartItems.FirstOrDefaultAsync(item => item.Name == product.Name);

// If the cart item doesn't exist, create a new one and add it to the cart context

if (cartItem == null)

{

cartItem = new Cart

{

Name = product.Name,

Price = product.Price,

Quantity = 1

};

\_cartContext.ShoppingCartItems.Add(cartItem);

}

// Otherwise, increment the quantity of the existing cart item

else

{

cartItem.Quantity++;

\_cartContext.ShoppingCartItems.Update(cartItem);

}

TempData["LoginSuccessMessage"] = "Product added to shopping cart!";

// Save changes to the cart context

await \_cartContext.SaveChangesAsync();

// Redirect to the shopping cart page

return Page();

}

When the user clicks the ‘Add To Cart’ button, the method triggers and retrieves the product based on the ‘productId’, It then uses the ‘FirstOrDefaultAsync’ method of the ‘\_cartContext.ShoppingCartItems’ property to retrieve the corresponding shopping cart item of matching name.

If no shopping cart item exists, it creates one, if one does exist, then increase quantity by 1.

The database is updated with the ‘.Update’ method with ‘cartItem’ as the parameter, changes are saved with ‘.SaveChangesAsync’ and a success messages is stored in ‘[TempData]’ once again to be printed. Finally, the code returns ‘Page();’ to reload the current page.

This code is present wherever the ‘Add To Cart’ button.

The ‘ShoppingCart.cshtml’ has behind code that works to display the added items from the ‘Cart.db’, it is functionally the same as the other codes that display from databases.  
  
private readonly CartDataContext \_cartDataContext;

[BindProperty]

public IList<Cart> Cart { get; set; }

public decimal TotalPrice { get; private set; }

public ShoppingCartModel(CartDataContext cartDataContext)

{

\_cartDataContext = cartDataContext;

}

public async Task<IActionResult> OnGetAsync()

{

Cart = await \_cartDataContext.ShoppingCartItems.ToListAsync();

TotalPrice = Cart.Sum(item => item.Price \* item.Quantity);

return Page();

}

The difference is the ‘TotalPrice’ property which is used to get the total price of all the items added, the ‘private set’ is to ensure no external code causes problems and inconsistencies with the property.

The total price is calculated by doing a ‘.Sum’ method which sums up an items price times quantity.

The ‘OnPostDeleteAsync’ for deleting items in a cart is modified slightly to account for quantity.

public async Task<IActionResult> OnPostDeleteAsync(int id)

{

var product = await \_cartDataContext.ShoppingCartItems.FindAsync(id);

if (product == null)

{

return NotFound();

}

if (product.Quantity > 1)

{

// If the product quantity is greater than 1, reduce the quantity by 1

product.Quantity--;

\_cartDataContext.ShoppingCartItems.Update(product);

}

else

{

// If the product quantity is 1 or less, remove the product entirely from the shopping cart

\_cartDataContext.ShoppingCartItems.Remove(product);

}

await \_cartDataContext.SaveChangesAsync();

TempData["LoginSuccessMessage"] = "Product removed!";

return RedirectToPage();

}

If the products quantity exceeds 1, then decrease the quantity by 1 and update the cart. The else statement completely removes the product instead. The changes are saved with an appropriate message to print.

public async Task<IActionResult> OnPostClearAsync()

{

\_cartDataContext.ShoppingCartItems.RemoveRange(\_cartDataContext.ShoppingCartItems);

await \_cartDataContext.SaveChangesAsync();

TempData["LoginSuccessMessage"] = "All items removed from the cart!";

return RedirectToPage();

}  
  
Finally, ‘OnPostClearAsync’ completely clears the cart with the ‘.RemoveRange’ command.

The shopping cart is not tied to users and is cleared after a log out.

**2.3.2 Search**

public async Task OnGetAsync(string searchQuery)

{

if (!string.IsNullOrEmpty(searchQuery))

{

Products = await \_productContext.Products.AsNoTracking()

.Where(p => p.Name.ToLower().Contains(searchQuery.ToLower()))

.ToListAsync();

}

}

The search bar uses this code in the ‘OnGetAsync’ method, if the string is not null or empty, then execute code to find products.

This is accomplished by retrieving the products from the database without keeping track of changes using the \_productContext.Products DbSet and the AsNoTracking method. The items whose names match the search query string are then filtered using a LINQ Where clause, and the results are returned as a list of products that are set to the class's Products field.

Of course the search page has constructors and properties initialised like other pages to get data from the necessary databases. The products also have an ‘Add To Cart’ button on each result, the code is the same as mentioned before.

**2.4 Miscellaneous**

This section details any extra features that cannot be categorised in the sections before.

* The products do not have an image property, the images and extra pages are manually set in the code.
* User role and the correct year for the trademark are shown at the footer.  
    
   @if (User.Identity.IsAuthenticated)

{

<p>Role: @User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Role)?.Value</p>

}

<p>Name: Rick Richard Harith</p>

<p>Email: T0321003@my.ntu.ac.uk || Phone Number: 07864676131</p>

<p>

Copyright Information: All images used are not copyright free. &copy; @DateTime.Now.Year

</p>

* If statements are used throughout the page to dynamically change the navigation bar for logged in users, non logged in users and admins.
* If statements were used to refer to certain products by Id to have a ‘More Info’ button, this button leads to the pages made in Assessment 1 for the products.
* Logging out causes the cart to clear with an appropriate confirmation message.
* Deleting users as an Admin also gives an appropriate confirmation message. The code is below:

onsubmit="return confirm('Are you sure you want to delete this user?');"

**3.0 Conclusions**

**3.1 Challenges Encountered**

The website does not have a session based shopping cart, if the databases were set up with foreign keys to tie to users to the shopping cart and products added, that would solve the issue.

The Add To Cart code is copy pasted a lot inefficiently, a partial view with shared class code would help make the code more neat and efficient.

Ideally, the product database should save images to display by saving it as binary data, this is not the case with the current website.

**3.2 Summary and Conclusion:**

The website meets most of the main requirements of the assessment and I am somewhat satisfied with the outcome. The website sets up three appropriate databases that save information correctly and reference the database for displaying products, logging in and sign up, shopping cart display and more. Using C# if statements with the Identity, Authentication and Authorisation system has enabled the website to be secure, so users must abide to certain conditions to access certain pages.

If I had more time, I would commit to making product images possible. The shopping cart should also be tied to sessions and specific users. Finally, the code should be more efficient and should use more shared model classes and controllers to handle certain repeating codes or input handling.

**Sources and References**

* C, tutorialsEU (2023). *C# Razor Pages - Complete Tutorial*. *YouTube*. Available at: https://www.youtube.com/watch?v=pmWFzzU\_NB4&t=873s [Accessed 29 Mar. 2023].
* Code with Julian (2021). *Beginners ASP.NET Core Identity Tutorial*. *YouTube*. Retrieved from: https://www.youtube.com/watch?v=5UfJeDcoC1k&t=95s [Accessed 16 March 2023].
* Dykstra, T., Likness, J. and Smith, J.P. (2022). *Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8*. [online] Microsoft.com. Retrieved from: https://learn.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-7.0&tabs=visual-studio [Accessed 15 March 2023].
* Digital TechJoint (2022). *ASP.NET MVC - How To Implement Role Based Authorization*. *YouTube*. Retrieved from: https://www.youtube.com/watch?v=qvsWwwq2ynE&t=196s [Accessed 17 March 2023].
* Learn Razor Pages (2021). *Working With A Model | Learn Razor Pages*. [online] Learnrazorpages.com. Retrieved from: https://www.learnrazorpages.com/razor-pages/tutorial/bakery/create-model [Accessed 14 March 2023].
* Ravindra Devrani (2022). *Upload image in dot net core APIs*. *YouTube*. Retrieved from: https://www.youtube.com/watch?v=dBoVlBhjkks&t=912s [Accessed 17 March 2023].
* Raw Coding (2022). *ASP.NET Core Authentication (.NET 7 Minimal Apis C#)*. *YouTube*. Retrieved from: https://www.youtube.com/watch?v=ExQJljpj1lY&t=378s [Accessed 16 March 2023].
* Saini, S. (2022). *ASP.NET 6 Razor Pages CRUD - .NET 6 Razor Pages Tutorial - ASP.NET Core Web App*. *YouTube*. Retrieved from: https://www.youtube.com/watch?v=3gthVb11Juc&t=7s [Accessed 16 March 2023].