Supply Chain Management System Report:

Coursework Title: CST2550 Group Coursework

Project Title: Supply Chain Management System

Group Name: MeAndTheOtherGuys.

Student Information:

- Name: Carlo Napolitano
- Role: Team Leader
- Student ID: M00812374

- Name: Fares Eisa.
- Role: Secretary
- Student ID: M00867008.

- Name: Omed Bahaddin
- Role: Developer
- Student ID: M00949847

- Name: Khalid A Aden
- Role: Developer
- Student ID: M00860524

- Name: Faisal Dahir
- Role: Tester
- Student ID: M00912132

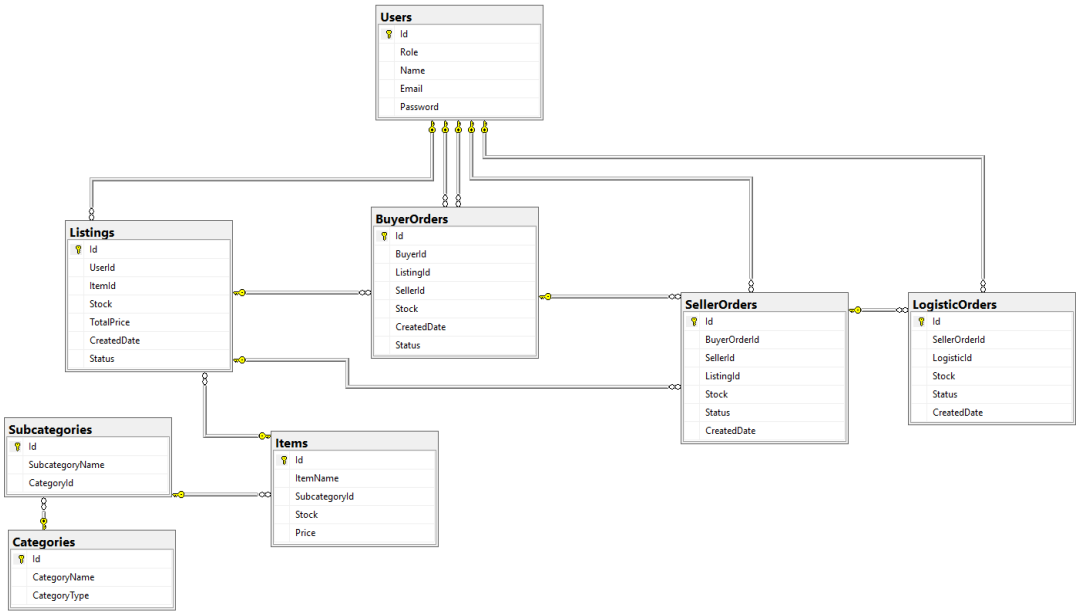
Submission Date: 18/04/2025

Introduction:

Following the coursework report content, I will be going in order, first starting with the introduction, design, Testing, and lastly conclusion. Whilst having a title change each time I am talking about a new section.

Supply management is a service provided to businesses that includes sourcing manufactured goods and managing inventory. We considered several service approaches, such as operating as a middleman allowing customers to search for items like "ropes" or "fibers," and connecting them with top, reliable suppliers while offering multiple purchasing options. Instead, rather than referring customers, we could produce and sell the items ourselves, using data on pricing and shipping we collected from research.

Design:



Entity	Connects To	Purpose
Users	Listings, Orders	Represents a platform user, can act as Buyer, seller, or Logistic.
Items	Listings	The actual product that is listed for sale.
Listings	Users, Items	A product listed for sale by a user with stock and price.
Buyer Orders	Listings, Users	Records when a user buys a listing.
Seller Orders	Buyer orders, Listings	Handles the seller's part of the transaction after a buyer order.
Logistic Order	Seller Orders, Users	Handles the logistics and delivery of the seller's order.

Selected Data structures:

One of the data structures we used was a List in our PageModels to store and manage collections of data that came from the database, which were then sent to the Razor Pages like ClientPage.cshtml and OwnerPage.cshtml.

```
2 references
public List<Category> Categories { get; set; } = new();
2 references
public List<Subcategory> Subcategories { get; set; } = new();
2 references
public List<Item> Items { get; set; } = new();
```

We used these lists to hold all categories, subcategories, and items fetched from the database. These are later used on the client dashboard to let users filter and view items they want to buy or sell. We went with

List because it doesn’t require us to know how many items there will be ahead of time, and it's super easy to use with LINQ and Razor's foreach.

```
<select id="subcategorySelect" class="form-select">
    <option value="">Select Subcategory</option>
    @foreach (var subcategory in Model.Subcategories)
    {
        <option value="@subcategory.Id" data-category-id="@subcategory.CategoryId">@subcategory.SubcategoryName</option>
    }
</select>
```

We used a list here so we could loop through all categories and display them as dropdown options. Since we're dynamically generating this list from the database, using a List made it easy to bind and render without any extra overhead.

```
    @foreach (var item in Model.Items)
    {
        <tr class="item-row" data-subcategory-id="@item.SubcategoryId" data-category-id="@item.Subcategory?.CategoryId" data-item-id="@item.Id">
            <td>@item.ItemName</td>
            <td>@item.Subcategory?.SubcategoryName</td>
            <td>@item.Stock</td>
            <td>@item.Price.ToString("C")</td>
        </tr>
    }
```

This is how we displayed items on the table. We used List<Item> so we could easily loop with foreach and render all items under the selected filters. Using something more complex like a dictionary would have made this harder to read and more complicated than it needed to be.

```
2 references
public List<Listing> Listings { get; set; }
```

This is used to store listings posted by users. Since we didn’t know how many listings each user would have, using a List made it easy to manage, display, and loop through them.

```
// ● Retrieve active listings for the item
var listings = _context.Listings
                       .Where(l => l.ItemId == item.Id && l.Status == "Active")
                       .OrderBy(l => l.Stock) // Order listings by stock, lowest first
                       .ToList();
```

We used .Where() to only fetch listings that were active and matched the item the user wanted to buy. This is way more readable and cleaner than writing a for loop with if conditions, and it works naturally with List.

```
    @foreach (var user in Model.Users)
    {
        <tr>
            <td>@user.Id</td>
            <td>@user.Name</td>
            <td>@user.Email</td>
            <td>@user.Role</td>
        </tr>
    }
```

We used foreach to loop through the user list and display them in a table. This keeps our code simple and avoids the headache of tracking indexes like you would in a normal for loop.

```
if (string.IsNullOrEmpty(UserName) || UserRole != "client")
{
    return RedirectToPage("/Login");
}
```

This check makes sure the user is logged in and has the correct role before they can access the client dashboard. It helps avoid crashes or unauthorized access, and we used if instead of exceptions because this is an expected check, not an error.

```
Subcategories = _context.Subcategories.Include(s => s.Category).ToList();
Items = _context.Items.Include(i => i.Subcategory).ToList();
```

```
BuyerOrders = await _context.BuyerOrders
                        .Include(bo => bo.Buyer)  // Include Buyer (User)
                        .Include(bo => bo.Listing)  // Include Listing details
                        .ThenInclude(l => l.Item)  // Include Item details in Listing
                        .ToListAsync();
```

We used .Include() and .ThenInclude() to load related data in one query. This saved us from writing separate queries for each related item and prevented performance issues from loops that make a DB call on every iteration.

Key Functionality's algorithms using pseudo code

This here shows the key algorithms' functionality for the razor page to operate, as well as the pseudo-code used to get a clear idea of how we want it to work and what exactly we want it to do. (to see all pseudo-code please click the pseudo folder on the GitHub page).

```
Client-Side Quantity Check
Source: Client.cshtml

Function toggleButtons():
    quantity ← Get quantity input
    If quantity > available stock:
        Disable buy button
    Else:
        Enable buy button
```

This front-end algorithm is used to prevent users from trying to buy more than what's in stock, If the quantity input exceeds the available stock for the selected item.

```
User Login and Role-Based Redirection:
Source: Login.cshtml.cs

Function LoginUser(email, password):
    user ← Find user where Email equals email

    If user is null:
        Return "Invalid login"

    If password is empty:
        Return "Password required"

    If user.Password is null:
        Return "User password missing"

    result ← VerifyHashedPassword(user.Password, password)

    If result is invalid:
        Return "Invalid login"

    Save user.Email, user.Role, user.Name in session

    If user.Role is "owner":
        Redirect to Owner Dashboard
    Else if user.Role is "client":
        Redirect to Client Dashboard
    Else if user.Role is "logistic":
        Redirect to Logistic Dashboard
    Else:
        Redirect to Home
```

This here handles verifying a user's email and password and redirects them based on their role. It pulls the user from the database using the entered email, checks if their password is correct using the PasswordHasher, and then stores the user information into a session state.

```
Register New User
source: Register.cshtml.cs

Function RegisterUser(name, email, password, role):
    If any existing user has Email:
        Return "Email already registered"

    If role is "owner" AND an owner already exists:
        Return "Only one owner allowed"

    If password is empty:
        Return "Password cannot be empty"

    hashedPassword ← Hash(password)
    Create new user with name, email, role, hashedPassword

    Save user to database
```

This here prevents account conflicts and handles basic input validation for the registration process.

Testing

Approach:

The testing we used was unit testing via the NUnit framework with the primary goal of verifying the individual components in the system — from domain models like User, Category, and Item, to page logic for client, login, logout, and owner interfaces. We used in-memory databases to mock data environments and applied data annotations and custom validations to simulate real-world input scenarios. The focus was on ensuring robust logic, data integrity, property interactions, valid input handling, and proper session and identity management for Razor Pages.

Table of Test Cases:

ApplicationDbContext Tests

Test Name	Purpose	Expected Outcome
#CanCreateDbContext	Verify context initializes properly	Context is not null
#DbSetsAreAccessible	Check DbSets like Users, Items exist	All DbSets are not null
#OnModelCreating_ConfiguresPriceColumns	Ensure Price properties are set up	Price columns are configured
#AddAndRetrieveUser	Add a user and pull it back	Retrieved user matches input
#AddAndRetrieveCategory	Confirm categories are saved correctly	Match between input and result

Category Model Tests

Test Name	Purpose	Expected Outcome
#DefaultSubcategoriesNotNull	Default subcategory list shouldn't be null	Subcategories is empty, not null
#AddSubcategoryWorks	Confirm subcategory can be added	Subcategories.Count == 1
#ValidCategoryPassesValidation	Valid data goes through validation	0 validation errors
#EmptyCategoryNameFails	Empty name throws a fit	≥1 validation error

Subcategory Model Tests

Test Name	Purpose	Expected Outcome
#CanAssignCategory	Assigning a category should work	Object assignment successful
#ItemsPropertyCanBeAssigned	Assign item list	Items.Count == 1
#ValidSubcategoryPasses	Valid input goes through cleanly	0 validation errors
#EmptySubcategoryNameFails	Missing name triggers error	≥1 validation error

Item Model Tests

Test Name	Purpose	Expected Outcome
#ValidItemPassesValidation	Valid item should validate just fine	0 validation errors
#EmptyItemNameFails	Missing name? Fail	≥1 validation error
#OverlyLongItemNameFailsValidation	Name too long? Fail	≥1 validation error
#InjectionStringValidatesIfWithinLength	SQL-looking strings are OK if not too long	0 validation errors

Listing Model Tests

Test Name	Purpose	Expected Outcome
#ValidListingPasses	All valid data, should pass	0 validation errors
#NegativeStockFails	Negative stock? Not today	≥1 validation error

User Model Tests

Test Name	Purpose	Expected Outcome
#ValidUserPassesValidation	Email, password, name all check out	0 validation errors
#InvalidEmailFailsValidation	Bad email format should fail	≥1 validation error

Client Page Tests

Test Name	Purpose	Expected Outcome
#ClientPageGet_PopulatesCategories	Ensure categories list is populated	Count == 1
#ClientPagePost_InvalidQuantityFails	Posting quantity = 0? Yeah, that should bounce	RedirectToPageResult
#ClientPagePost_ValidDataAddsListing	Valid listing data gets saved and calculated	Listing exists with correct TotalPrice

Index Page Tests

Test Name	Purpose	Expected Outcome

#IndexPageLoads	Just checks it loads without a meltdown	No exceptions thrown

Login & Logout Page Tests

Test Name	Purpose	Expected Outcome
#LoginWithValidCredentialsRedirects	Good credentials = smooth login	RedirectToPageResult
#LogoutSignsOut	Logs you out and bounces back to homepage	SignOutAsync called, redirects to /Index

Owner Page Tests

Test Name	Purpose	Expected Outcome
#OwnerGet_PopulatesUsers	Loads users from DB	Users.Count == 1
#OwnerGet_PopulatesListings	Also confirms listings populate correctly	Listings.Count == 1

Conclusion:

We have completed a supply chain management web application that allows businesses to both sell and buy manufactured goods they need. The platform also allows users to view all ongoing listings they are involved in, whether they are currently buying or selling. Additionally, it makes it easy for the owner to manage the entire system, as they can view all registered users, track every item being sold or bought, and see who is buying from or selling to whom.

One issue we ran into early on was deciding which framework to use since there were a lot of options and most of them were new to us. To figure this out, we tried a few different ones to see which would work best. One of the first frameworks we used was a console app. It worked well for handling backend logic and gave us no major issues at that level. However, once we tried integrating it with a frontend, we ran into problems. Some parts of the backend didn't connect properly with the front end, and that's when we decided to start overusing a different approach. We switched to Razor Pages, which gave us a much smoother experience. It was a lot easier to manage communication between the front end and back end, and everything felt more connected. But even with Razor, we still ran into issues. For example, when using AJAX to fetch data from SQL, the entire web page would refresh unexpectedly, which made it hard to support multiple interactive buttons on the same page. We eventually figured out how to solve these issues, but by the time we did, the original structure of our project had started to break apart. After understanding where things went wrong and learning what worked best, we restarted the project one last time. This time, everything came together — we knew exactly what to do and how to build it properly, and we were finally able to get everything running smoothly without any major issues.

In the future, we would approach a similar task by spending more time researching each framework and understanding its pros and cons beforehand. This would help us make more informed decisions early on and save time, rather than learning things the hard way during development.

(Conclusion)