Realisation – Supermarket FreshChoice BV case

Authors: Roko Mladinić, Daniella Namuli, Simeon Markov, Zed Minabowan & Kristiano Mizher

Introduction

Commented [ZM1]: @Minabowan,Zed P.X.O.

This document describes the realization process of the Supermarket FreshChoice BV case. In this document we'll be talking about what we did to create our final product. We'll also go in depth about how our product is structured.

Project structure

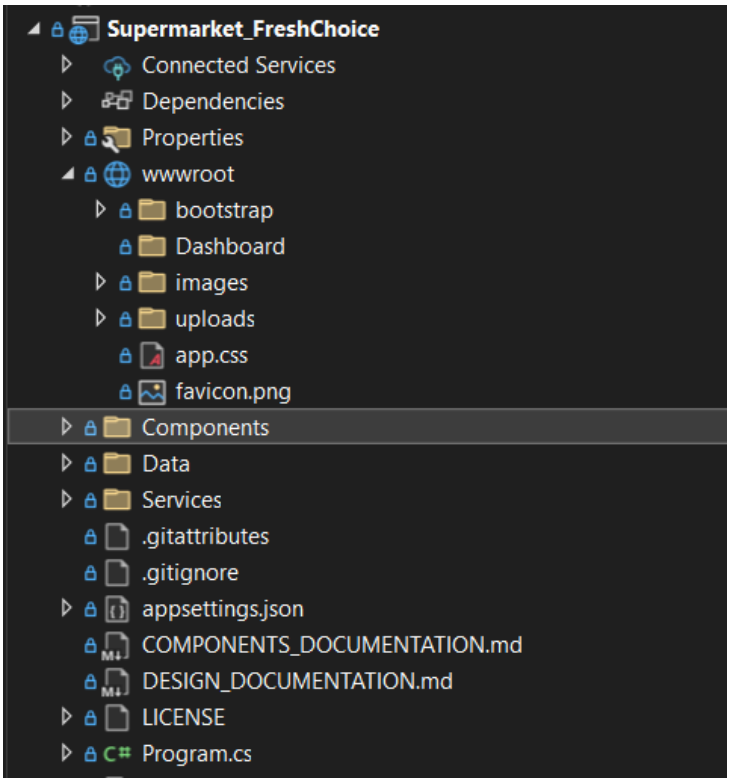
Commented [MS2]: @Markov,Simeon S. Task: Complete the project structure, database structure

This section explains the architecture of the project and usability.

System architecture: C# & Blazor (Server-side rendering), MVP (Model-View-Presenter) three-tier architecture.

Backend: Almost entirely on C# with some JavaScript. The Framework handles the routing; entities are defined under model classes, code-first approach.

Frontend: HTML, CSS, Bootstrap.

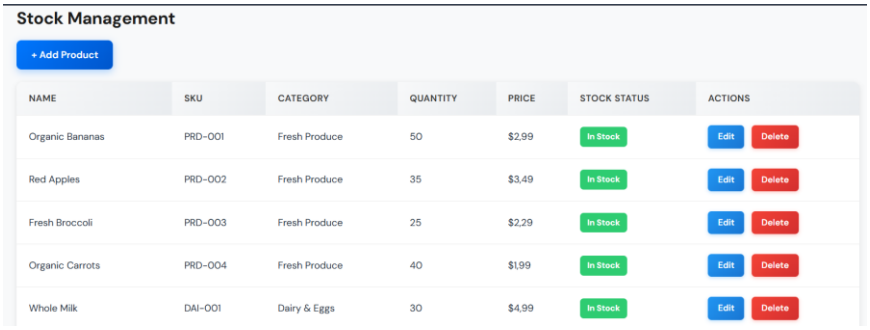


Design vs Realisation

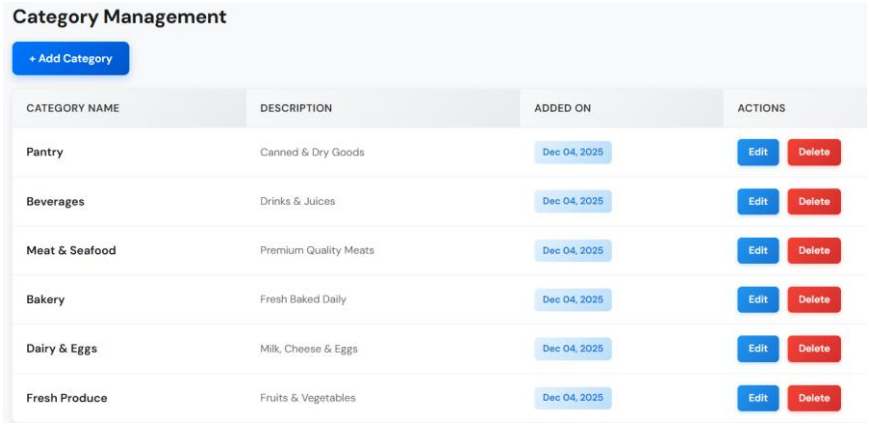
Commented [MS3]: @Markov,Simeon S. Task: explain the changes

Changes in initial design

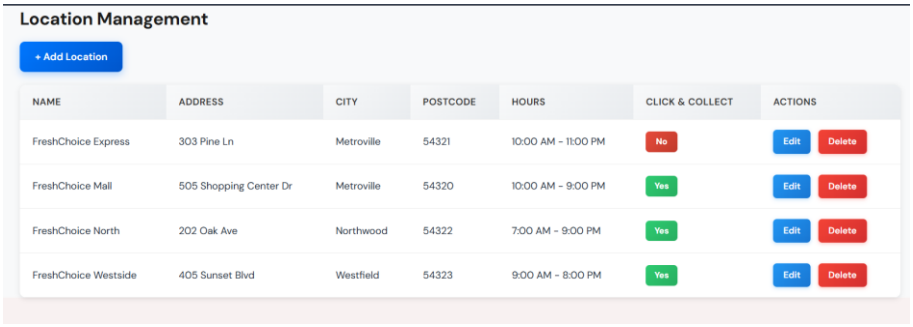
Stocks page: The page was modified to include several other fields like price, category as well as action buttons were added on the side for easier access (e.g. edit, delete):



<u>Category page:</u> The category page was added with relationship (many-to-one) towards the Stock resource, keeping the same layout and colors as the stocks page:



<u>Location page:</u> Currently the page resembles the rest of the pages in terms of design (data presented in tabular format with quick access buttons for editing and deleting), but the 'search by postcode' feature is not implemented.



Database changes: New additional tables were added to the initial database design schema. The following tables were added:

Locations: Used for storing the data for the location of each market, which have one-to-one relation with the Employee entity (One employee could be present (working) only in one location).

ShiftSlots: Used to keep track of the different shifts each employee has in a certain market's location.

Employees: Keep registered employees with their credentials like name, address, role...

Code documentation

Commented [MS4]: @Markov,Simeon S Task: Document key code snippets

Main logic: The code structure is MVP (Model-View-Presenter) that is coming by default from the framework Blazor (it could be extended to MVVM, but this is not the case here). Since it is a component-oriented framework, the practice of using reusable components is kept. The Model part is responsible for defining the business data, the View for the frontend and the Presenter serves as a bridge between Model-View, it handles the backend, user input and returns a response.

Models: Classes (real-world objects) are defined under models, which are later translated to data objects by the ORM, the result of which is new tables in the database with all their columns and constraints if any:

```
public class Category
{
    [Key]
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string? Description { get; set; }

    [Required(ErrorMessage = "The Date is required.")]
    public DateTime AddedOn { get; set; }
}
```

```razor
@page "/locations"
@rendermode InteractiveServer
@using Supermarket_FreshChoice.Data
@using Supermarket_FreshChoice.Data.Models
@using Microsoft.EntityFrameworkCore
@inject ApplicationDbContext DbContext

<PageTitle>Location Management</PageTitle>

<div class="title">Location Management</div>

<div class="top-bar">
    <button class="add-btn" @onclick="OpenAdd">+ Add Location</button>
</div>

@if (showModal)
{
    <div class="modal-overlay" @onclick="CloseModal">
        <div class="modal-content" @onclick:stopPropagation="true">
            <div class="modal-header">
                <h3>@(isEditing ? "Edit Location" : "Add New Location")</h3>
                <button class="close-btn" @onclick="CloseModal">&times;</button>
            </div>
            <EditForm Model="location" OnValidSubmit="Save">
                <DataAnnotationsValidator />
```

Explanation: This is a partial snippet from the Location component page, showing the defined route (/locations) and database injected into that component. Then there is the conditional rendering which checks the state of the component (it is in modal state or not? If so show create functionality or edit if not and there is available data). The UI is built with the help of Bootstrap and custom CSS.

```razor
private async Task Save()
{
    if (isEditing)
    {
        var existingLocation = await DbContext.LocationArea.FindAsync(location.Id);
        if (existingLocation != null)
        {
            existingLocation.Name = location.Name;
            existingLocation.Address = location.Address;
            existingLocation.City = location.City;
            existingLocation.Postcode = location.Postcode;
            existingLocation.OpeningHours = location.OpeningHours;
            existingLocation.OffersClickAndCollect = location.OffersClickAndCollect;
            DbContext.LocationArea.Update(existingLocation);
        }
    }
    else
    {
        await DbContext.LocationArea.AddAsync(location);
    }

    await DbContext.SaveChangesAsync();
    await LoadData();
    CloseModal();
}
```

Explanation: The image above is a part of the Location component, where the code is inside the component page (View-Presenter relation). This approach is suitable for small-scale rapid development applications. The logic is first to check the state (it is in create or edit state?). If it is in edit state: assign the new value to the old value of that property and if it is in create mode: fill all the required fields, validate and save to the database. Saving is an asynchronous operation to prevent handling several instances of creation at the same time.

Development methodology and techniques

Commented [ND5]: @Namuli,Daniella D. finish this section

Styling Conventions

- **Scoped CSS**
  Each page has its own .razor.css file so styles don't mix and every component stays clean.
- **BEM-like naming**
  We use clear class names like .Stocks so the code is easy to understand.
- **Responsive design**
  The pages are made using media queries so the layout works on all screen sizes.
- **Consistent spacing**
  We use rem units to make spacing scales properly across different devices.

Practices Implemented

DRY (Don't Repeat Yourself)
We avoid repeating code by reusing components and keeping shared styles in one place.

Component Design

- **Single Responsibility**
  Every component has one job, making it easier to maintain.
- **Composition**
  Bigger UI parts are created by combining smaller components.
- **Configurability**
  Components can accept parameters, so we don't need to rewrite them.
- **Type Safety**
  We use `[EditorRequired]` for important parameters to avoid mistakes.

Data Flow

- **One-way binding**
  This is used when data only needs to be shown.
- **Two-way binding (@bind)**
  This is used for inputs where the value needs to be updated in real-time.
- **EventCallbacks**
  Components communicate smoothly with each other using events.
- **Async operations**
  Data loading is done with async methods to keep the UI fast.

Performance Techniques

- **Lazy loading**
  Data is only fetched when needed, which improves the load time.
- **Conditional rendering**
  We only render certain UI parts when required.
- **Entity Framework optimization**
  We use `.Include()` to load related data efficiently.

Accessibility

- **Semantic HTML**
  We used the correct tags like `<nav>` and `<button>` to improve accessibility.
- **ARIA attributes**
  These were used when needed to help screen readers understand the UI.
- **Alt text**
  Images include alt descriptions for better accessibility.

Error Handling

- **Try-catch blocks**
  These prevent the application from breaking during async operations.
- **Loading states**
  This is shown while data is being fetched, so the user knows what is happening.
- **Empty state messages**
  This is displayed when there is no data available.
- **Console logging**
  This is used during development to track issues.

Conclusion

With the implementation phase complete, we have successfully transformed our initial wireframes and requirements into a tangible, working application. Our focus now shifts from realization to the essential process of verifying and validating the product to guarantee reliability and user satisfaction.

Commented [MR6]: @Mladinić,Roko R. me