* Introduction

With Smart home systems on the rise the desire for a fully integrated home system would be a nirvana For the CMP3000 modules project, the decision to make an integrated home database system for the use of everyone would be a great place to start. The main functionality to this app is to modernise the archaic and less efficient ways of keeping stock of what’s in the home, seeing what recipes a person can make and what is needed to be bought at the shop with a shopping list. There are many applications out in the world which give a user a great collection of recipes, but the functionality stops there, the application doesn’t tell you if you have all the ingredients or tell you where they are. The purpose of the “Stokk” application is to integrate home and errand life and simplify all the tasks by having one application that has a wide range of functionality, with the capability to share peoples recipes from around the world and add your own.

Since the application was being made to be more efficient and useful for the everyday home user, then a mobile application would be chosen for development, this makes perfect sense as there is a large range of people in the world with access to as smartphone, and so statistically more users can be reached if mobile development was followed.

The initial plan was to create a Xamarin front end for a mobile application, but due to complications throughout development with Xamarin forms and compatibility, a different architecture was chosen. The desire to be accessible on a mobile was still key so the use of a web application where not only mobile users can use it but those who can access the web page, this widens the possible range of users.

* Background, objectives & deliverables

The initial idea of the project came from the increase in smart home technologies and the automation of home life, the project scope initially included consuming online shopping or receipt reading API’s to help the user initially start their database, but due to time constraints and scope creep the application was kept simpler for base functionality. This application is a steppingstone in the right direction towards automated home life, although now a user must input all the data, the functionality of the shopping list and recipe handling is still demonstrated.

Objectives

To create an application that increases efficiency in the home

Diagram

Description automatically generated

Business case

To make an application that makes the home life i.e. cooking and shopping more accessible and integrated.

To make an application that helps out with home life by integrating smart home technologies and database handling.

User stories

As a user I want to be able to choose a meal from the ingredients I have

As a user who has chosen the meal, I want to be told where the ingredients are and how much is needed

As a user I wish to make my own meal recipe

As a user I wish to make my own recipe through editing someone else’s

As a user I want a to be able to look at my shopping list and see what items are getting low and what items are needed indefinitely

As a user I want to be able to create my ideal stock so the app can tell me how many of that item I need to get when shopping

As a user I want to be able to log in and out of the system easily so they can check their stock quickly

As a user when I am low on an item, I want to be notified

As a user I want the system to update after choosing a meal

As a user I want to be able to customize my storage space to be personalised to my home

Diagram

Description automatically generated

The initial login page functionality is almost the same for both login in and signup, although here I could either decide to either have another pop up modal for the input of information, but for simplicity sake to start with the user will just input the values into the email and password and then if they click sign up the account will be created.

A picture containing text, map, sky, indoor

Description automatically generated

The navigation page will be the main page which controls the flow to other pages of functionality, here we would want a simplistic design which is always giving they user the ability to go back and revert their previous action (HCI principle), that is why on every page the user should be able to refer back to the previous page, just in case the user made a mistake, so on the main page the user can log out which will direct them to the login page.

From the navigation page all main functionality pages are located here, the user can go to meals, shopping list, storage, and settings (maybe profile page).

Diagram

Description automatically generated

Ability to go back to the previous page.  
checking the requirements are sated before adding item to save errors occurring.

Most functionality will take the user to the following page and once the user wants to interact with that page say add an item to the shopping list, this will come up to the user as a pop up modal as to stop constant redirects to constant pages, once finished easy to close and the user is on the previous page.

Diagram

Description automatically generated

Later in development after realising the application would be using modals, added in another activity which showed the application closing the modal window as to reach initial activity.

Diagram

Description automatically generated

Diagram

Description automatically generated

The initial activity didn’t have the modal closing

A picture containing text, sign, screenshot, sale

Description automatically generated

This was the initial ERD Which was conceived early in the project design phase, although this is ideally how the final database design would look like (without the correct relations), a simpler form of the application was built initially to try and build up the functionality, with a simple table that contains the Item with its relevant information such as quantity and the location of the item by room and cupboard attributes.

Then it was important to make sure that each item was connected to a user so that a user could see their own storage spaces as well as items within them, the next step was to Connect the user table and storage table where Username is a foreign key that is attributed to each item.

Graphical user interface, application

Description automatically generated

Login page, user must input username and password for both log in and sign up.

When user presses login or sign up, the inputs from the username and password, and checks them along the database to see if its either already in there and has the correct values and if it isn’t it can be added to the database. user is then taken to main page.

Diagram

Description automatically generated

This page will be the first thing a user sees when they first log in, this page works as a hub to direct to other pages.

Each button on the main page navigates to the other pages of functionality.

Sign out button takes user to Login page and signs user out.

Graphical user interface, text, application

Description automatically generated

Main page button to take user back to main page directory.

On the meal page a user can view meals that they can make, much like a recipe book.

This is separated out into meals they can currently make with the ingredients they have.

And the other section being all the other meals, where the user will be told what ingredients they need to get in order to make.

Each meal once pressed will open a pop-up modal that shows information about the meal, the ingredients needed, how it’s made, and some extra info.

User can add a meal and share it with other users of the application so that anyone can make any variant of meals with their own spin.

Graphical user interface, text, application, chat or text message

Description automatically generated

The modal will come over the current page for the user to interact with where the user will see the ingredients involved and how it’s made.

If the user presses find ingredients the user is given another pop-up modal which shows each of the ingredients and where they are positioned.

Once the user has found the ingredients the user will be taken back to this recipe page where they can make the meal under the instructions.

Once the meal is complete the user then presses recipe complete where the database will take away the items used in the meal away from the database.

Possibly here is where we could ask the user how they would rate the meal (maybe even delay 30 minutes after they did the meal ask what they rated).

Graphical user interface, text, application, chat or text message

Description automatically generated

Here the application will tell the user where each of their ingredients they need for the meal is positioned, once the user has found the items ready for the meal, they click the back to recipe button where they can see how to make the meal.

Graphical user interface, text, application

Description automatically generated

The storage page will show the user each of the spaces where they store their ingredients.

Each storage space is separated into which room they are in.

Once a storage space is clicked a pop-up modal showing which ingredients are in that storage space is shown.

The user can add a storage space.

Graphical user interface, text, application, chat or text message

Description automatically generated

The modal for the storage space will come over the storage page showing which ingredients are in that space with quantity and weight shown as well.

User can add an item of their own accord to the current space in the database.

Storage space one will go here

Graphical user interface, text, application

Description automatically generated

Most of the shopping list will automatically be done within the database, but user can view the list as well as add or remove items.

Graphical user interface, text, application

Description automatically generated

The add item modal will be used in a few instances, for adding items to a cupboard space and adding items to a shopping list.

* Literature review (if applicable)

Reviewing other applications both webs based and mobile, to gain features and critiques.

For research into the area of recipe handling, both web-based, and mobile applications were analysed to reach a broader range of possible functionality and obtain inspiration for user interfaces within the similar areas.

For the Web based websites both bbcfood.com and Goodfood.com were studied.

Graphical user interface, website

Description automatically generated

**Pages features**

-on the main page of BBC food, it shows some suggested possibly seasonal meals, with some choices to navigate to other selections of recipes. They have a nav bar at the top (which won’t be necessary for mobile application as there will be the main navigation page that does the same job).

**Speculative functionality**

Could use a feature of suggested meals either by popularity or by season, this would help users get meals which are deemed as better (user can still use their own recipes which they input).

A rating system for meals could be possible asking after the user has finished what they thought of the meal and what they would rate it on a scale of 1-10. Alternatively, following how many people used it and how many people wanted to save it to favourites. Could even make a percentage of how many people added to favourite over how many people have used it (possibly even who viewed it as well)people added to favourite over how many people have used it (possibly even who viewed it as well)

One of the applications most like Stokks vision was paprika, which is a mobile application which is an app that does recipe handling as well as storage control and groceries. This is one of the only other applications that encapsulate all the functionality I want to integrate into the application, some features may be left out due to scope creep and some features will have a different design method behind it.

Graphical user interface, application

Description automatically generated

The first page the user is directed to on the paprika application is the recipe page, as the main functionality of paprika is a recipe handling. The user is able to add recipes manually on this page and then view them and be able to navigate through them.

Graphical user interface

Description automatically generated

Instead of going for a main page design paprika elects for a side navigation bar to go from page to page.

Which gives quick easy access to the user without having visual clutter of different page buttons, only showing when the user requests.

Graphical user interface, text, application, email

Description automatically generated

The next page is to partner the first page of recipes, here the user is able to search online on an in app browser, for recipes, with some suggestions and links to food recipe sites, her the user can attempt to copy and paste the recipe manually with clipboard or try the download feature which will save the recipe straight to recipes.

Alternatively stokk will use a mass database of everyone’s meal recipes, both the users personal recipes and other users will be stored, so everyone will be able to try anyone’s recipe, and there the recipe can be rated either through whether the users went through with making the meal or ask for the user to give a rating. This rating will then be able to display the meals popularity so on the meals page it can show popular meals, and the users’ personal meals as a priority, and allow the user to search through all the possible meals.

Graphical user interface, application, Teams

Description automatically generated

The edit recipe page is similar to the design of how stokk will be designed, although the user will not be able to rate the meal here. Although there could be 2 rating systems, one for the users personal rating of meals and the publics rating which everyone sees, and the recipes will be sorted in. possibly give user option to sort by personal rating or public rating.

A picture containing shape

Description automatically generated

Paprika’s pantry feature is similar to that of stokk’s storage system, the key difference being paprika has just one page showing all of the ingredients in the house, whereas stokk will separate everything out by room and cupboard, this coincides with the recipe functionality which tells the user where the ingredients for a certain meal are.

* Method of approach
* Legal, social, ethical and professional issues
* Project management

Every 2 weeks a meeting with a project manager/ tutor was held where every session was attended and a stand up took place, although a far cry from the daily stand up meetings which are used in the agile methodology in the work environment, this helped with vocalising issues and answering any queries one had.

On the alternating weeks a small blog post detailing what we did that week was also written for the tutor to follow.

Although these meetings were helpful extra precautions were put into place to help this project move forward in a more structured manner. Bi weekly meetings with another tutor were held, this helped keep the application stay on track towards the end as these were used to make sure the correct functionality was being focused on instead of features and gold plating that wasn’t necessary earlier in development.

For the final month or so of the project Daily stand-up messages were sent to a person who was keeping track, sent at the end of the day they outlined what progression was made during the day, what difficulties were had and what the plan for the next day was, this helped structure the working days on a day to day basis rather than a fortnightly, so progression was tracked more closely. This helped drastically with the progression of the project.

Pre sprints:

-Where the idea was initialised and formed

-Initial erd and attempted normalisation

- researched other applications

-created repository

-did research on technologies

- mocked up UI

-some tutorials on api and xamarin

- attempted to make xaml pages

-create statements for tables written

Attempted sprint one for a month

First coding sprint 2 weeks

Ignore User functionality for the moment can show other functionality without it

Adapted and changed architecture and technologies to web asp.net

Tutorials for Web forms mvc and asp.net api took place

Test environment using the technologies but with simple data one table

Fully CRUD

Redesign of table structure based on Household

Created simple database missing recipe and shopping list functionality:

User, Household, Room, StorageSpace, HouseHoldItem, Item and Item Type

Created and published api

Create front end app to receive endpoints and manipulate data Full CRUD

Created filter pages

Created tables for shoppingList/item and recipe/item

Due to broken API, rebuilt connecting to all tables

Front end created for the tables with filter pages

Initiation

research

Design phase 1

Tutorials

Redesign architecture

Tutorials

Practice environment

* sprints

Initially I had desired to start the project off by creating user functionality (login and sign up functionality) but due to having numerous issues with the architecture an architectural redesign had to take place, where most functionality is based on Household instead of individual users. So instead of staying on the user functionality as the first sprint, the goal of having a test environment where I could practice the languages and test the connections within the architecture. This consisted of creating a simple web page that displays an index of the storage table, along with its data, here a user can CRUD the data. This helped practice the skills needed and to test if the application could work.

so after The test environment was created the first coding sprint could take place, within the first coding sprint a base start for the applications were created where a front end MVC application communicates with a published api application using API endpoints to change data in a mostly finished database structure.

Also, some mock adapters were created in the API to mock a connection to a barcode and shopping basket API.

Fully CRUD with filter pages filtering Item by Item Type, Rooms by Households, Household Items by Household (this is to show ingredients in a given household), storage space by room, Shopping list item by Shopping list (This displays an individual households shopping list) and recipe Item by Recipe(To show Ingredients for a recipe).

* End-project report
* Project post-mortem
* Issues

During the development for the application some issues arose which needed to be resolved so the project could continue.

One of main visions of the project was to make the application mobile friendly, thus why Xamarin was chosen, as its intuitive. One of the key issues was that Xamarin was not working as expected. previously when using Xamarin, all the work had been done within the university, in a Smeaton lab that has access to Xamarin and the necessary libraries. Due to COVID access to the lab was hindered. The assumption was that the computers that I had access to would be capable of coding in Xamarin like the lab, but there were continuous issues following through. On the main desktop where the project was originally created, due to having an AMD processor the computer was not able to run the emulator to run and test the front end. To try and resolve this issue a laptop with an intel CPU was used, this got passed the issue of being able to use the emulator, but another problem arose, where no matter what version of NuGet packages was used none would ever install correctly making functionality come to a standstill. As progression was halted another solution had to be put in place, so instead of using Xamarin for the front end, a web based front end was considered. Even though the initial plan of the application was to make it mobile accessible so although the application won’t be able to be on the application store, the ability to access the application on any device that can access the web will increase the amount of possible users able to access the application. To make sure the user is still getting the experience that we desired the application will be scaled for mobile phones, so the UI is still user friendly to those using phones with smaller screens.

After having decided a Web based front end was best, many practice projects were used to both go through the tutorials of ASP.Net web forms, MVC and asp.net API as well as folders where I applied these practices to my project. To start with Web forms was attempted, but after failing to display a simple set of data from an API endpoint an attempt using the MVC architecture by utilising the view pages (razor pages cshtml) was attempted. This attempt at displaying the data from the api endpoint onto the web page was the base for a test environment where I could deploy the application, CRUD the data and make sure the data is sent to the database with some exception handling (based on the database exceptions).

**Architecture design**

Using Microsoft SQL server management studio, database tables were created with Query statements.

SQL Queries for Create Tables

USE COMP3000\_SSharpe

GO

CREATE TABLE [User]

(UID int NOT NULL,

CONSTRAINT AK\_UID UNIQUE(UID),

CONSTRAINT PK\_UID PRIMARY KEY NONCLUSTERED (UID),

FriendlyName varchar (50)NOT NULL,

[Password] varchar (50)NOT NULL,

Email varchar(50) NOT NULL

);

GO

CREATE TABLE HouseHold

(HHID int NOT NULL,

CONSTRAINT AK\_HHID UNIQUE(HHID),

CONSTRAINT PK\_HHID PRIMARY KEY NONCLUSTERED (HHID),

HHName varchar (50)NOT NULL,

AdminUser int NOT NULL,

CONSTRAINT FK\_AdminUser FOREIGN KEY (AdminUser)

REFERENCES [User] (UID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

RestrictedUsers int,

CONSTRAINT FK\_RestrictedUsers FOREIGN KEY (RestrictedUsers)

REFERENCES [User] (UID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE Room

(RID int NOT NULL,

CONSTRAINT AK\_RID UNIQUE(RID),

CONSTRAINT PK\_RID PRIMARY KEY NONCLUSTERED (RID),

RoomName varchar (50)NOT NULL,

HHID int CONSTRAINT FK\_HHID FOREIGN KEY (HHID)

REFERENCES HouseHold(HHID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE StorageSpace

(SSID int NOT NULL,

CONSTRAINT AK\_SSID UNIQUE(SSID),

CONSTRAINT PK\_SSID PRIMARY KEY NONCLUSTERED (SSID),

StorageName varchar (50)NOT NULL,

RID int CONSTRAINT FK\_RID FOREIGN KEY (RID)

REFERENCES Room (RID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE ItemType

(ITID int CONSTRAINT AK\_IYID UNIQUE(ITID),

CONSTRAINT PK\_ITID PRIMARY KEY NONCLUSTERED (ITID),

ITName varchar (50) NOT NULL

);

GO

CREATE TABLE Item

(IID int CONSTRAINT AK\_IID UNIQUE(IID),

CONSTRAINT PK\_IID PRIMARY KEY NONCLUSTERED (IID),

ItemName varchar (50) NOT NULL,

ITID int CONSTRAINT FK\_ITID FOREIGN KEY (ITID)

REFERENCES ItemType (ITID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE HouseHoldItem

(HHIID int CONSTRAINT AK\_HHIID UNIQUE(HHIID),

CONSTRAINT PK\_HHIID PRIMARY KEY NONCLUSTERED (HHIID),

QTY int NOT NULL,

SSID int CONSTRAINT FK\_SSID FOREIGN KEY (SSID)

REFERENCES StorageSpace (SSID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

IID int CONSTRAINT FK\_IID FOREIGN KEY (IID)

REFERENCES Item (IID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

HHID int CONSTRAINT FK\_HHoldID FOREIGN KEY (HHID)

REFERENCES HouseHold(HHID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE ShoppingList

(SLID int NOT NULL,

CONSTRAINT PK\_SLID PRIMARY KEY NONCLUSTERED (SLID),

HHID int CONSTRAINT FK\_HouseholdID FOREIGN KEY (HHID)

REFERENCES HouseHold (HHID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

);

GO

CREATE TABLE ShoppingListItems

(SLIID int NOT NULL,

CONSTRAINT PK\_SLIID PRIMARY KEY NONCLUSTERED (SLIID),

SLID int CONSTRAINT FK\_SLID FOREIGN KEY (SLID)

REFERENCES ShoppingList (SLID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

IID int NOT NULL

CONSTRAINT FK\_ItemID FOREIGN KEY (IID)

REFERENCES Item (IID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

ITID int CONSTRAINT FK\_ItemTypeID FOREIGN KEY (ITID)

REFERENCES ItemType (ITID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

QTY int NOT NULL

);

GO

CREATE TABLE Recipe

(RecipeID int NOT NULL,

CONSTRAINT PK\_RecipeID PRIMARY KEY NONCLUSTERED (RecipeID),

Details varchar (200) NOT NULL,

Instructions varchar (MAX) NOT NULL,

UserID int CONSTRAINT FK\_User FOREIGN KEY (UserID)

REFERENCES [User] (UID)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

GO

CREATE TABLE RecipeItem

(RecipeIID int NOT NULL,

CONSTRAINT PK\_RecipeIID PRIMARY KEY NONCLUSTERED (RecipeIID),

Item int CONSTRAINT FK\_RecipeItemID FOREIGN KEY (Item)

REFERENCES Item (IID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

RecipeID int CONSTRAINT FK\_RecipeID FOREIGN KEY (RecipeID)

REFERENCES Recipe (RecipeID)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

QTY int NOT NULL,

Weight int

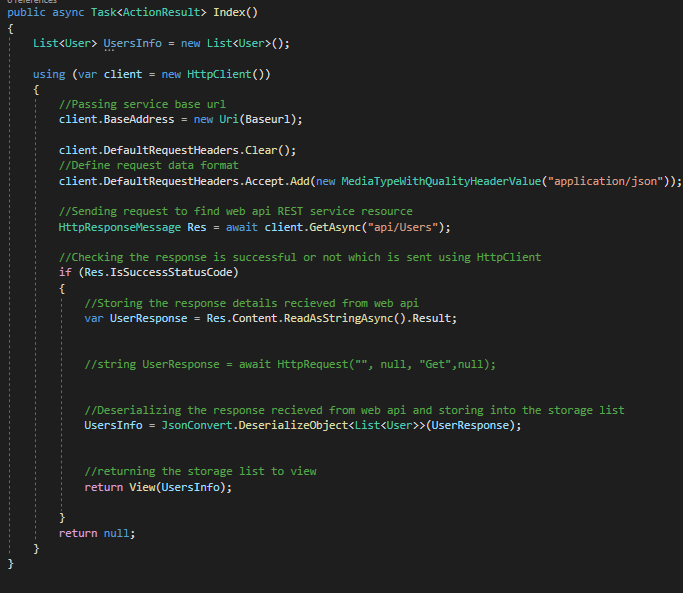
);

GO

The API was created by using ASP.NET framework web application, The API Application is in the architecture of MVC, where the Models hold the shape of the data from the desired tables, so for each table in the database there is a model to accompany it, these were created by adding an ADO.Entity Model . The API application has a set of API controllers which Create the API endpoints for all the Gets Posts Puts and Deletes, these are autogenerated from the wizard, choosing the data entity model that each controller works off of, these controller classes derive from the API Controller class rather than the Usual controller class used by MVC applications. The Views in this application didn’t have much use apart form a home page where it can direct to the raw data (xml), of each API GET endpoint.

This API Application was published so a separate Front end application could be made in parallel to receive the API endpoints and consume/ Push the data to and from the database through the API using the URL “web.socem.plymouth.ac.uk/FYP/SSharpe/” as a base address, this helped as testing could be done using the endpoint instead of localhost, so forticlient didn’t have to be used, as well as not having to run 2 applications at once all the time for testing.

The desire to keep confusion to a minimum and abide by the separation of concerns a separate application was created for the front end, the front-end application is an ASP.NET MVC web application, it has sort of the same structure as the API application apart from the fact that the Model doesn’t derive from an ADO entity model, and the controllers don’t derive the API Controller class. This means that the model is just a mirror of the Model.cs file within the API application, where it has the same attributes as the table with gets and sets within each. The controllers in the MVC application are Used to control the pages and the functions within them, by calling the API endpoints for the relevant page. Although the Views were pretty much ignored in the API application, the front-end application will be dealing with the User, so the views were utilised. After creating a controller, the IDE gives you a folder which has the same name as the controller, this folder holds the cshtml files for each page/controller. So within the User Folder there are many cshtml files which were created using the wizard where each table has an index view, Edit, Create, Details and delete cshtml page. Within the User Folder There are Methods for each page and functionality, which all interact with the API endpoints.



using FrontEndMVC.Models;

using Newtonsoft.Json;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Threading.Tasks;

using System.Web;

using System.Web.Mvc;

using System.Diagnostics;

Each Controller needs to use these libraries and classes to function.

The base address “web.socem.plymouth.ac.uk/FYP/SSharpe/” sits at the top of all the MVC controllers. In figure… the index function Initialises a List with User datatype this will hold all of the data from the API call and in the correct format.

List<User> UsersInfo = new List<User>();

Then it initialises a new http client by using the “using” action word.

using (var client = new HttpClient())

Within this using Http client, we give the client a base address by passing the base URL.

client.BaseAddress = new Uri(Baseurl);

Define request data format, the data being dealt with is Jason so

client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

Send a request to find the web API REST service resource, as this is a get function client.Getasync is used where the string parameter notates the desired endpoint so for index users, the base url is web.socem.plymouth.ac.uk/FYP/SSharpe/ so within the Get async its parameter will be the string “api/Users” to reach the desired endpoint.

HttpResponseMessage Res = await client.GetAsync("api/Users");

The function then checks res to see if it reached success status code, if not the function returns null, if success then.

Store the response details received from the api into UserResponse using readAsStringAsync function.

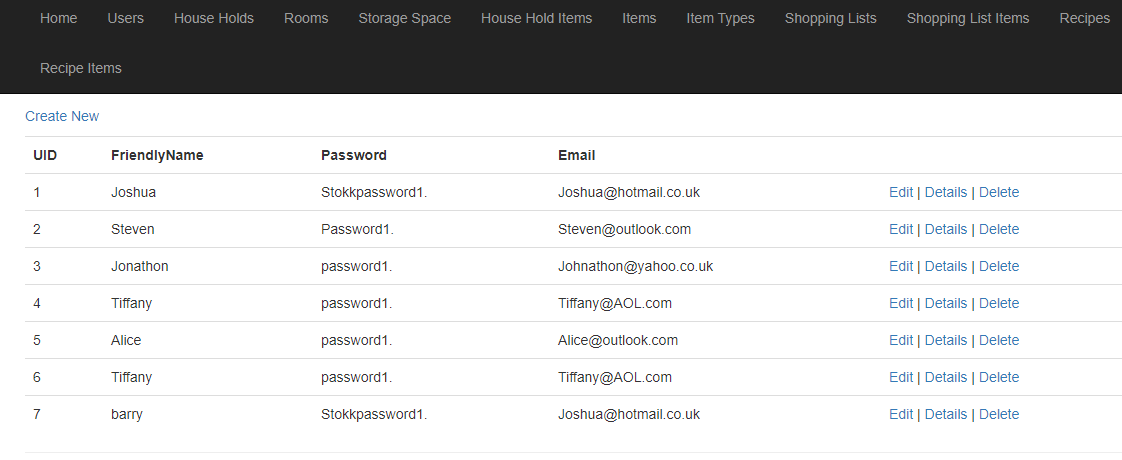
var UserResponse = Res.Content.ReadAsStringAsync().Result;

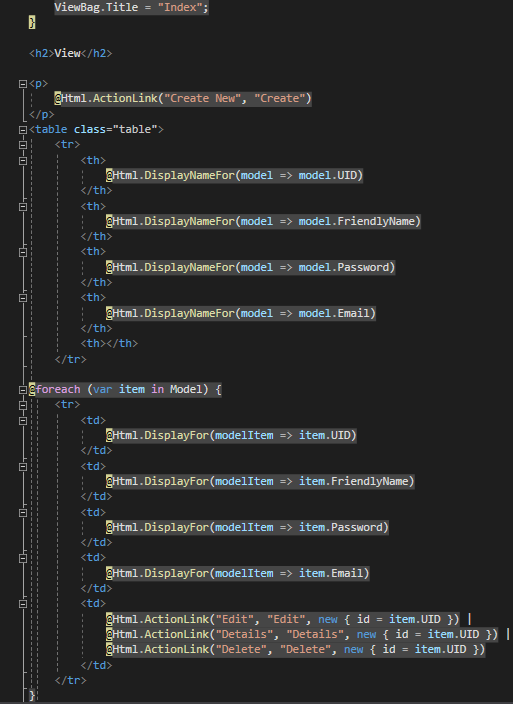
Deserialize the response received from web API and storing into the list using Json.Convert and deserializeObject

UsersInfo = JsonConvert.DeserializeObject<List<User>>(UserResponse);

Then return the user List to view

return View(UsersInfo);



This is the Index Function within the Users Controller, this is used when looking at any of the collections of the tables on the index pages (this one for the user), it displays all values in a table which is based off of the model within the MVC application. At the bottom of the cshtml page you can see the actionlinks for other pages, here they show name of the button, and what page to be directed to along with a parameter to pass across, so for instance if a user clicked on the details button, the user would be taken to the detail page where UserID is passed as a parameter so the details of a singular user can be seen. 

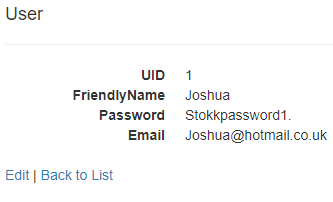
Most Get Functions are similar in structure to the index function, although instead of a List the datatype of the model is used so instead of returning a list of Users it returns the information of a single user

User UserInfo = null;

So instead the GetAsync function uses (“api/Users/” + id ), this function was called DetDeserialize where id is a parameter.

HttpResponseMessage Res = await client.GetAsync("api/Users/" + id);

This function is used on many pages and functions, Get Users Details, Get Users edit, Get Users Delete

All relocate to the desired page passing the id of an individual user across to the delete details and edit pages, the only other GET that doesn’t use GetDeserialize is the create User page, as data doesn’t need to be passed to the page, it only needs a blank template to send the information across. 

All post functions don’t require the 2 lines using client.DefaultRequestHeaders, as you are not receiving any data only posting.

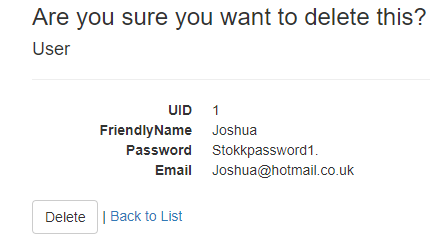
Create user is a normal Post function, that converts the body to Json format sends it to the api, and then redirects back to the index page after

var postTask = client.PostAsJsonAsync<User>("Users/", user);

Graphical user interface, application

Description automatically generated

Delete User although a delete Async, the cshtml page passes the information in post format so [httpPost] needs to be declared before the function, but still uses DeleteAsync, that redirects back to the index page after.

var deleteTask = client.DeleteAsync("api/Users/" + id.ToString());

Edit User takes in the parameters of user for the body of changes and id to refer to the user’s id and endpoint. Yet again due to the way cshtml pages send data this function had to be denoted as [httpost]

Even though it uses putAsJsonAsync, which also serializes the input for the API endpoint.

var putTask = client.PutAsJsonAsync<User>("api/Users/" + id, user);

Graphical user interface, text, application, email

Description automatically generated

Certain controllers have a filter function and page. For instance, the items table can be filtered by ItemType using the function FilterByIT. where the actionLink in the index cshtml page calls the function FilterByIT and passes the Item type ID as a parameter, then redirects to the filter page (Which is in the list format like the index page).

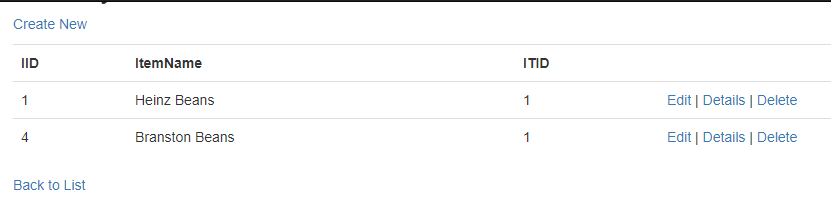
The function is similar to the Index function up until the last which instead of just returning the ItemInfo list will filter it where item.ITID (itemID) is the same as the id passed through as a parameter.

return View(ItemInfo.Where(Item => Item.ITID == id));

A picture containing table

Description automatically generated

So if the user clicked the filter by it button on the first item, the ITID of the generic beans which is 1 is passed to the filterByIT function where it returns the Filter list page where Items ITID == 1.



Once pressed page displays all items under the item type of beans which is Heinz beans and Branston beans.

To simulate different API’s being connected to the system, a folder was created in the API application called Adapters, this folder holds for each desired connection there is a c# interface class that defines the adaptor’s interface and a class that conforms to the interface, along with some adaptations to how the model is formatted.

For example, there is a dummy barcode adapter and interface named ImpBCDUmmy and IBarcodeAdapter respectively. The Ibarcode interface defines item findproductByBarcode passing barcode as an int parameter.

Item FindProductByBarcode(int barCode);

Within the Model, Item.cs, this code was added so the data being passed conforms to the model structure, where it can be passed no parameters, parameters of string or id, or those and TypeID.

public Item() { }

public Item(int ID, string Name)

{

IID = ID;

ItemName = Name;

ITID = null;

}

public Item(int ID, string Name, int TypeID)

{

IID = ID;

ItemName = Name;

ITID = TypeID;

}

}

ImpBCDUmmy class creates a dictionary collection of fake data called BCValue, this dictionary has integer and item data being passed to it. Fake data was input where the first number of the dictionary is the id of the barcode that it receives from the external Barcode API, and in the second section of the dictionary a new item is created with item name and id being given for that item.

IDictionary<int, Item> BCValue = new Dictionary<int, Item>() {

{ 88887777, new Item(1234567, "Beans")},

{ 77776666, new Item(2345678, "Sausages")};

Have a class that returns the data in the format of the datatype Item

private Item ReturnTranslate(object data)

{

return (Item)data;

}

Have another class called findProductByBarcode that takes int item as a parameter, it then returns the ReturnTranslate function where BCValue[item] is the parameter.

public Item FindProductByBarcode(int item)

{

return ReturnTranslate(BCValue[item]);

}

Another similar dummy adapter was also created for receiving shopping from a basket online, where instead of looking at the Item table it looks at the HouseHoldItem table.

For each of these dummy adapters an api endpoint was created, to use the adapter;

private IBarcodeAdapter barcodeAdapter = new ImpBCDummy();

had to be instantiated at the start of the main.

Making sure that the Response type is the Item data type, calls the GetBarcode function.

[ResponseType(typeof(Item))]

public async Task<IHttpActionResult> GetBarcode(int barCode)

{

return Ok(barcodeAdapter.FindProductByBarcode(barCode));

}

So if using postman or browser, the endpoint:

“http://web.socem.plymouth.ac.uk/FYP/SSharpe/api/Items?barCode={88887777}”

Displays the dummy data related to that barcode lookup which is beans.

Functional requirements change, found the idea of telling the user where the items are before a meal unnecessary as most people know where their items are.

Issue with determining how much the user used and keeping track of it, system is flawed if the user has to input every time.

* Conclusions
* Reference list and bibliography