Technical Delivery

Enterprise Computing Team Project

CA472

Name: Jamie Behan
Student Number: 18406986
Email: Jamie.Behan6@mail.dcu.ie

Name: Bernard McWeeney
Student Number: 18384466
Email: Bernard.McWeeney2@mail.dcu.ie

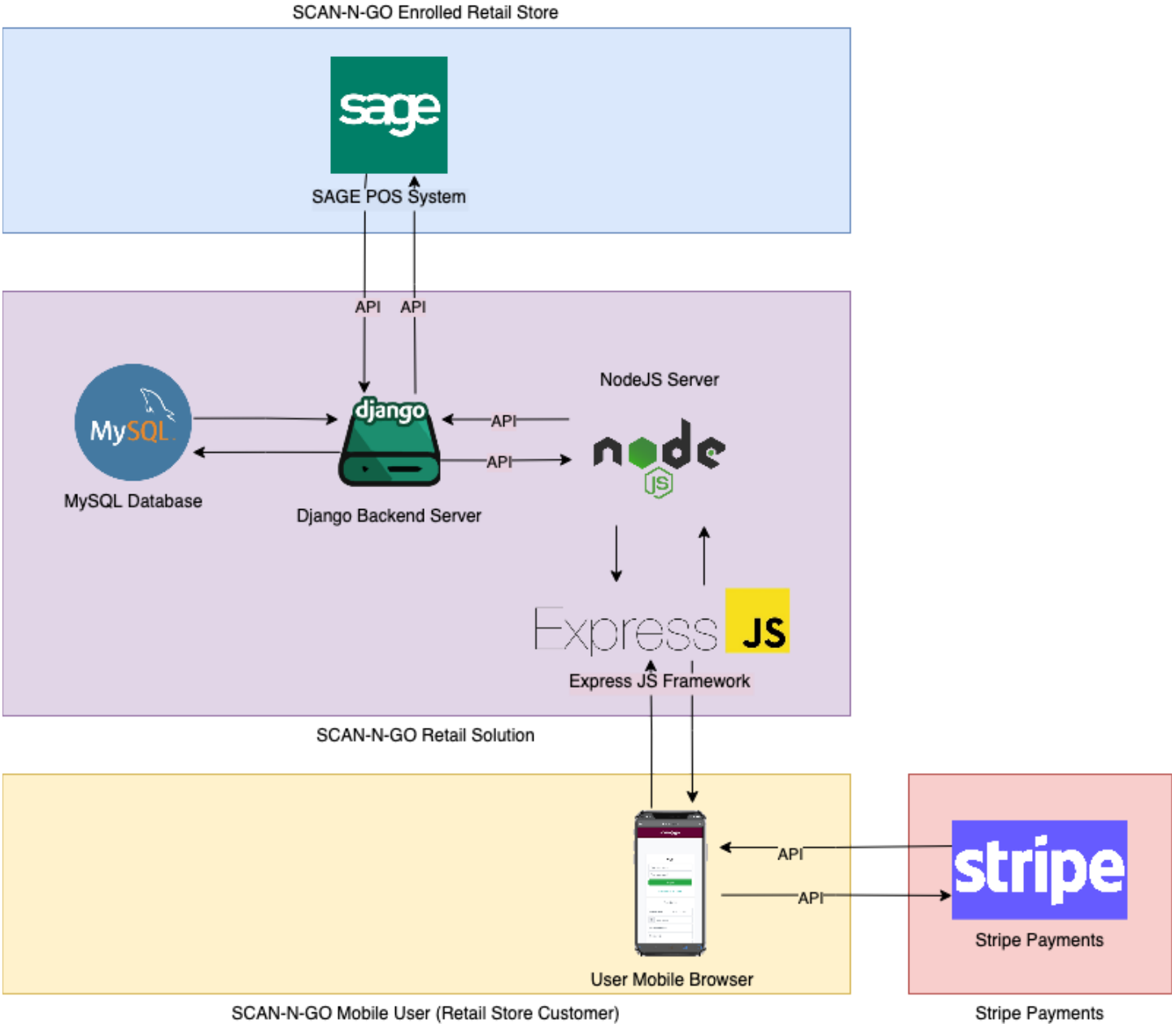
Name: Shaun Kee
Student Number: 18308546
Email: Shaun.Kee2@mail.dcu.ie

Submission: 19 Apr 2022
Programme: EC4
Module: CA472

Table Of Contents

1. Software Architecture

SCAN-N-GO's Software architecture is similar to the architecture used to run an e-commerce system alongside a bricks and mortar retail store due to the system's dependency on API communication with retail stores POS for managing stock levels and creating business documents and the usage of an external payment gateway to take payment. Within the SCAN-N-GO system we have a DJango backend + NodeJS Frontend because it is a widely used and well documented method of developing sophisticated web applications.



MySQL Database

Our MySQL database will be our primary database. It will hold all the tables shown in the Logical Data Model. As we have opted for MySQL rather than the Django default of SQLLite (due to storage limitations of SQLLite) MySQL will have to run on its own dedicated server.

Django Backend Server (Running Django REST Framework)

Our backend server will be the backbone of our web app. It will be the component that will compute all logic for the running of SCAN-N-GO. The server will retrieve requests from the front end (via API call) and return a response from the MySQL Database.

User's Mobile Browser

The Client's web browser is where they will physically interact with our platform. The client's browser will render the pages provided by the ExpressJS framework and will call on the backend API through a modern UI.

NodeJS

The NodeJS server will serve web pages to the user.

ExpressJS

ExpressJS allows for the dynamic filling of web content on the pages that nodeJS will serve to the clients browser.

Stripe Payments

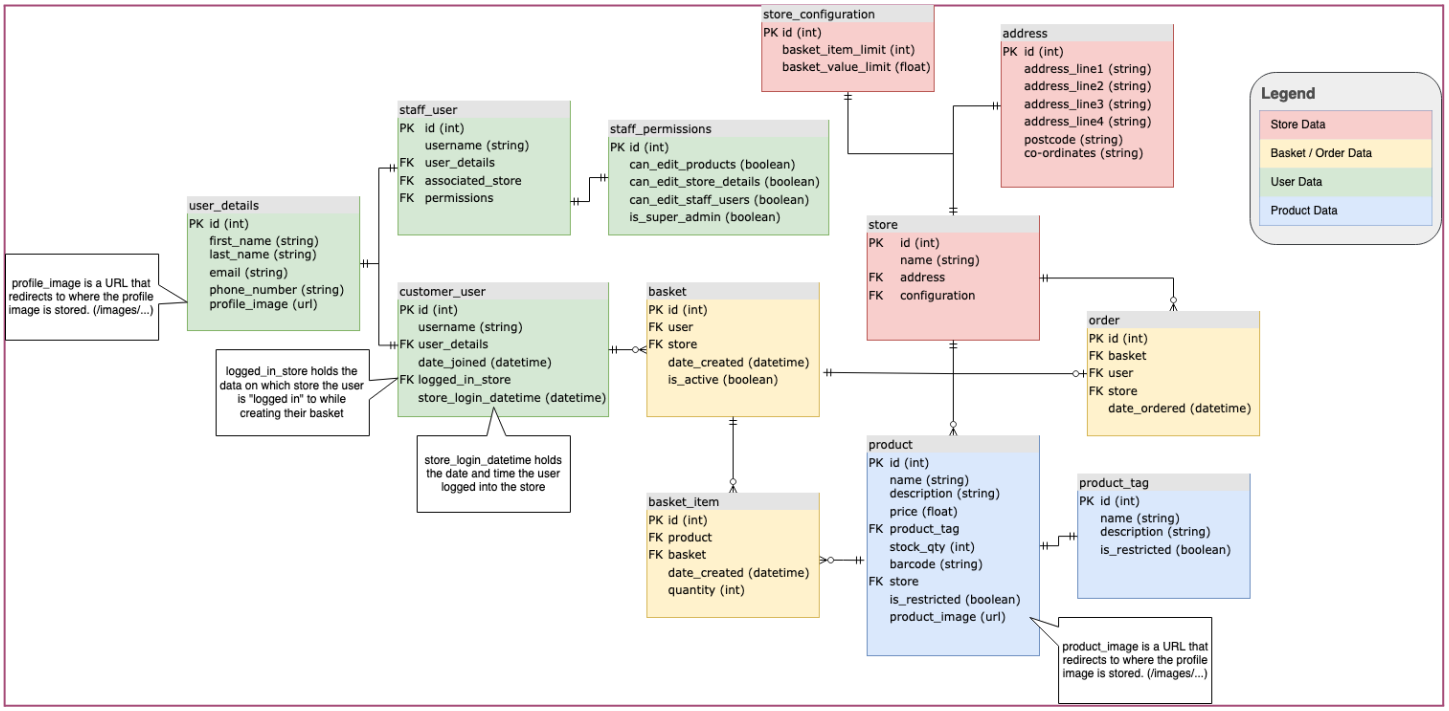
Stripe payments is a 3rd party payment processor. The retail store will open an account with Stripe, and we will use Stripe's API to send customers' payment details for processing. An authorised or unauthorised payment response from Stripe will then trigger the frontend to either checkout and complete the users order or display an error message.

Sage POS System

For this example we will use SAGE as our integrated POS (Point of Sales) system. SAGE is one of the most used retail software packages in the world. SAGE retail software creates transactions, tracks stock levels, and much more. We will communicate with the retail store's SAGE instance to add transactions and update stock levels when orders are placed and also update SCAN-N-GO's internal stock records to make sure SAGE's stock levels and SCAN-N-GO's stock levels are consistent.

2. Database Design

Our data will be stored on our dedicated MySQL database which will be running on its own dedicated server instance on Microsoft Azure. Since Microsoft Azure provides us with ***Scale Up*** functionality, we will be able to store a total of 150TB of data on this database before we'll have to consider moving to a different database management system. Since we have passed the responsibility of physical data storage to Microsoft Azure (PaaS) we will not have to worry about any data storage requirements other than maintaining the database. All sensitive data such as passwords will be SHA256 encrypted to ensure our users' accounts are secure and safe.



See bigger view of diagram in **[Appendix 2]**

Store Data

All SCAN-N-GO enrolled stores are represented as a line in the ***store*** table in the DB. Each store has a configuration which is the store's SCAN-N-GO '*settings*', this information comes from the **store_configuration** table. Each store also has an address which is stored in the **address** table.

Basket / Order Data

Each **basket** can contain any amount of **basket items** which take a **product** and quantity as its data. This is generally the way e-commerce stores store basket data.

User Data

A user can be a **customer_user** or **staff_user.** Both instances of users have user details which comes from the **user_details** table.

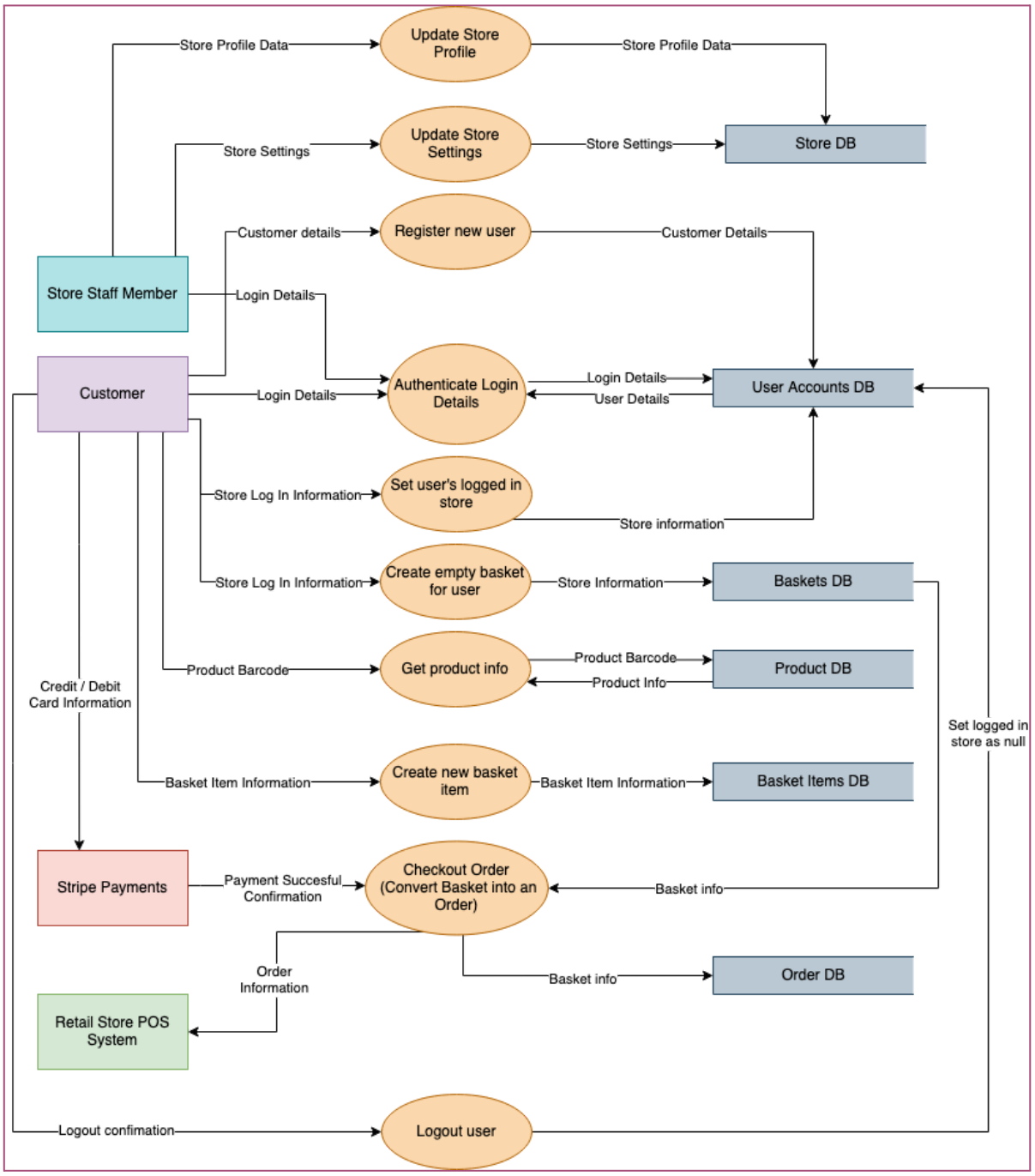
Product Data

Each product has a product tag to categorise the product. The tag information is stored in the **product_tag** table.

3. DFD & UML Diagrams

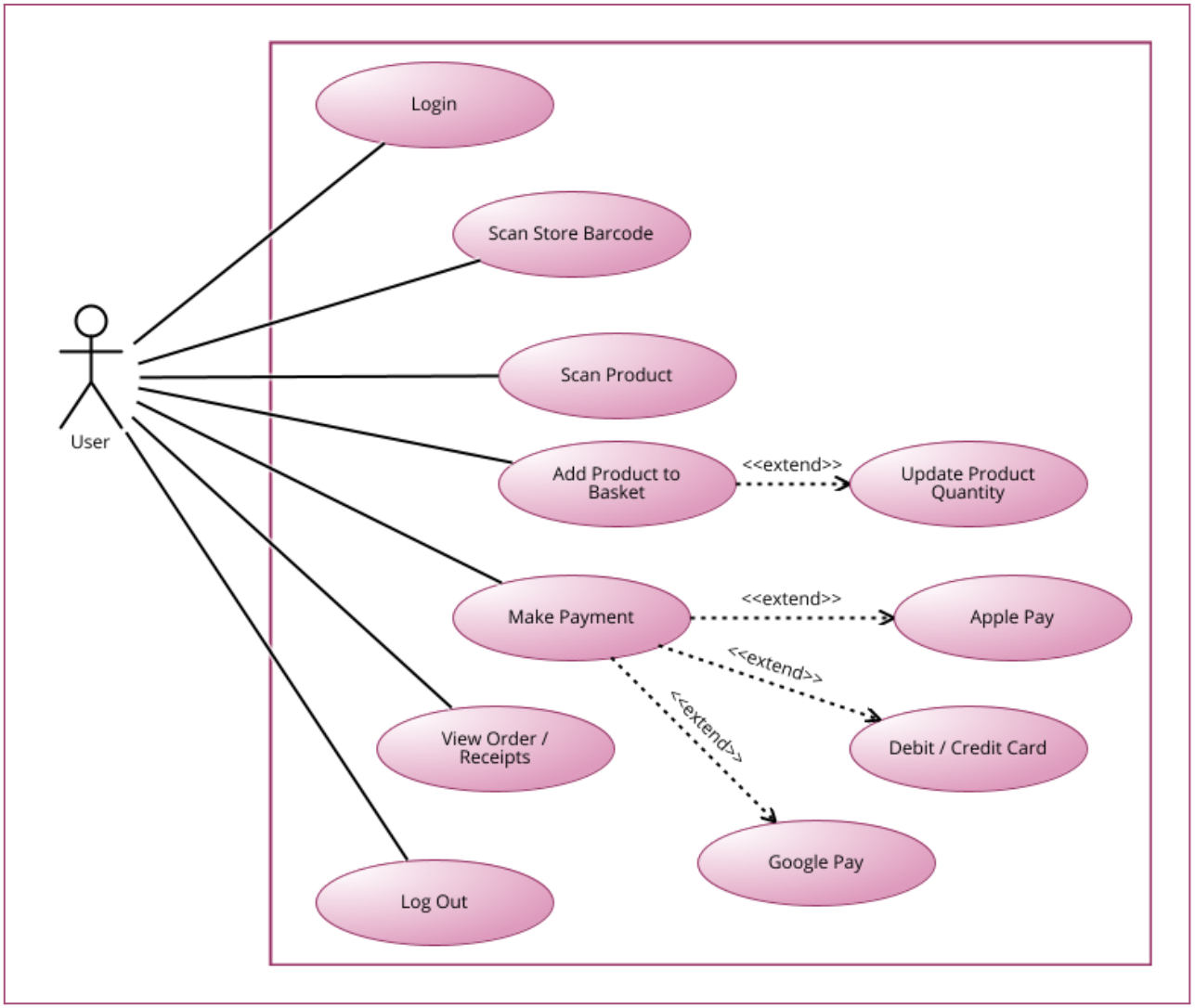
Data Flow Diagram

For our Data Flow Diagram we have created a Level 3 DFD with Yourdon & Coad notation. This Data Flow Diagram shows how all information and data flows through the SCAN-N-GO system for both the Store staff members accessing and using the Admin System and also the retail store customers accessing the standard web application's front end. Our external entities are Store Staff Members, Customers of Retail Stores, Stripe Payments & the retail store's Point of Sales system (POS System). In this case, each table in the DB is represented as a data store and each process is represented by an orange circle **[8].**



Unified Modelling Language Diagram

User UML Diagram



Login

- The User logs into the system.

Scan Store Barcode

- Once logged in, the user can then enter any store with SCAN-N-GO technology and scan the store login barcode.

Scan Product

- Once the user has scanned into the store, they can then scan any barcode product within the store.

Add Product to Basket

- When the user scans a product they will then be prompted to choose the quantity of the item they wish to purchase and add it to their cart.

Make Payment

- Upon the user reaching their basket they can make a payment for their scanned goods. SCAN-N-GO has multiple payment options - Apple Pay, Google Pay, Debit/Credit Card.

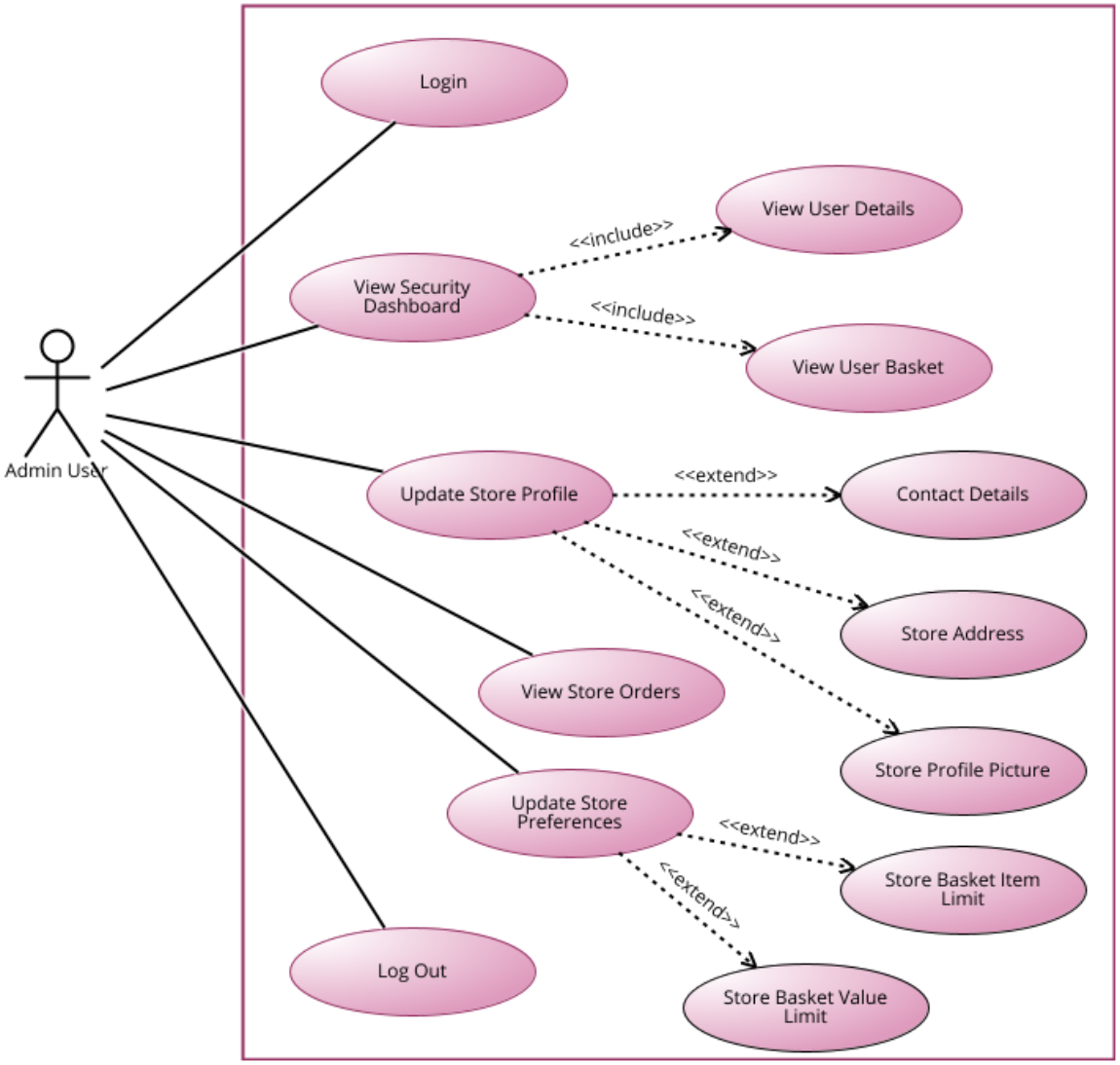
View Order / Receipt

- Once payment has been completed, the user is presented with a receipt which they can view.

Log Out

- The user can then log out of their account having purchased their goods in-store.

Admin UML Diagram



Login

- The Admin logs into the system.

View Security Dashboard

- Once logged in, the Admin can then access and view the security dashboard. From here, user details and each user's basket are displayed to the Admin.

Update Store Profile

- The Admin user can update the store's profile, details such as contact details, store address and store profile picture can be altered.

View Store Orders

- The Admin has the ability to view all previous orders / purchases made by customers within the store.

Update Store Preferences

- The Admin user can update their store-specific preferences in relation to limiting the amount of items a user has in their basket as well as placing a monetary limit on customers baskets'.

Log Out

- Once an Admin user has logged into the Admin dashboard they have the option to log out at any time.

4. External software interfaces

SCAN-N-GO Technology relies on three external software interfaces to provide the SCAN-N-GO service. We require one external interface for processing payments, one interface for the scanning and reading of AZTEC & EAN-13 barcodes and finally one interface for updating transactions and stock levels within the retail store's POS System.

External Payments Processor - Stripe, Inc

Processing payments is a fundamental service that SCAN-N-GO will be unable to provide without the aid of a third party. For this, we have decided to go with Stripe Payments because they are the market leader in helping startups process payments and they are incredibly up to date with 3DS2 (3D Secure V2), unlike most Irish providers of payment processing. To process payments we will use Stripe's HPP (Hosted Payment Page) to deliver the payment details securely to Stripe, this means our system will never have to store or even see sensitive information from the customers. Once the payment information is submitted to Stripe, our frontend will await a response of either ***Authorised*** or ***Unauthorised.*** This response will simply trigger the checkout process or an error message to the user, making it easy to integrate and maintain.

Barcode Scanner & Reader - SCANDIT Technology

Due to the limited timeframe to develop the SCAN-N-GO platform, we decided to outsource our barcode scanner to SCANDIT Technology rather than developing our own bespoke barcode scanner. For SCANDIT Barcode Picker to work, we import the SCANDIT SDK for Web Javascript Files and configure the scanner in a HTML `<script>` tag. We can then call on the scanner to open on command and trigger Javascript functions once barcodes are detected. There are however cons to this method of barcode scanning as SCANDIT will eventually charge us for using their software and it has a limited amount of configuration available to us. Initially we tried to use QuaggaJS barcode scanner in our software which is open source and has a wealth of configuration possibilities. We did however run into cross compatibility issues while we were testing on different phone platforms so in the end the only logical choice of scanner provider was SCANDIT. In the future we plan to experiment with developing our own barcode scanning technology to cut out this third-party service provider.

Retail Store POS Interface - SAGE Group plc

As an example for the project demonstration, we will demonstrate how we would integrate with SAGE POS systems. SAGE POS is one of the retail sector's most used retail systems and SAGE also provides a large amount of API's to communicate with the SAGE System. We are aware that there are many different providers of POS systems but we understand that all major providers provide APIs to communicate with their system and that integration with SAGE would not be majorly different to integrating with any other POS System. Once an order is created with SCAN-N-GO, the Django Backend Server would communicate

with SAGE's `/sales_invoices` API to create the Sales Invoice and transaction for the sale. Once the sale transaction is created on SAGE's system, the SCAN-N-GO backend server would call the `/stock_items` API to alter the stock level for that product. Once those two API calls are made, the sales process would be complete on SCAN-N-GO's side and also the retail store's POS side, meaning that stock levels and sales transactions would be up to date on the retail store's system, without any manual intervention. To allow our system to easily "plug-in" to any retail POS system and its APIs we will develop a graphical user interface for configuring a store's SCAN-N-GO instance which will make it easy to route API calls from the Django backend server to the retail store POS system. Obviously, during the initial introduction of SCAN-N-GO in the Irish market, we will be unable to provide our service to a retail store without the said retail store having a POS System that has API's that allow communication between the SCAN-N-GO backend server and the POS System. However, we believe that if our rollout of SCAN-N-GO Technology is a success, we will be able to upsell our SCAN-N-GO enrolled stores a SCAN-N-GO POS System and eventually penetrate the Irish market with a complete SCAN-N-GO retail system (Incl. POS).

5. Performance Requirements

SCAN-N-GO has two customer segments, customers of fast-paced retail stores and owners of fast-paced retail stores. The performance requirements of these differ from one another.

Customer of Retail Stores

Security

The security aspect of SCAN-N-GO's application is an essential performance requirement for customers of retail stores as this customer segment will be making purchases via the application using their banking details. To ensure this is handled effectively SCAN-N-GO has incorporated Apple Pay and Google Pay integration into the application. As well as this, Stripe handles all inputted debit and credit transactions. By using these services, it takes advantage of large infrastructures with a core focus on security.

Usability

SCAN-N-GO is designed to allow customers to enter a retail store, scan barcoded goods, swiftly make the payment and then leave the store. To further enable the effectiveness of SCAN-N-GO's mission, the application must make sense to the user. It must handle user error effectively, follow UI/UX guidelines or principles and be aesthetically pleasing in order to increase the overall user experience and usability of the application.

Functional Suitability

Aforementioned, it is of the utmost importance that the users can use the application in a way that makes sense to the user. To accompany this, the application must be functionally correct. The application should be designed appropriately to fulfil the value proposition for the user. The application should be well optimised, the scanning process should be intuitive and the checkout process should be quick and effective.

Owners of fast-paced Retail Stores

Reliability

The Admin dashboard is an important component for the retail store. It displays useful information to the staff of the store such as the number of SCAN-N-GO users that have scanned into the store, the number of items in their baskets alongside which items are within their baskets and more. It is important the Admin dashboard is reliable and works as intended throughout the day. If it is deemed to be unreliable in terms of performance and doesn't have proper fault tolerance or recoverability it defeats the purpose of the Admin dashboard.

Usability

The Admin dashboard is designed to aid the retail staff and provide a quick and simple overview of information at any given time. The Admin dashboard should be designed for a fast-paced environment which allows for quick actions and access to information with ease. The overall usability of the system is an important factor for these reasons.

Functional Suitability

Once again, to support previously discussed points. The Admin dashboard must be functionally complete, it must have an array of features that aid the in-store staff and not hinder them. All information should be available at a glance. An important aspect of this system is that there is not an overload of information for the user which could potentially subtract from the suitability of the system.

6. Technical Challenges

SCAN-N-GO recognises that there are some technical challenges that are required to be overcome to allow for SCAN-N-GO to have a viable place in the market. These are outlined below:

Integrating with Store Point-Of-Sales Systems

For us to operate in a retail store, we need to know what products the store has and the quantity of these products. A technical challenge for us is integrating with a retail store's point of sale system so that we can have access to the store's product database. We need access to the store's product database to know the available products in store, the price of the product, promotions assigned to a product and the quantities available. We also need to have access to update the product quantities, if a user has purchased a product using our app.

Similarly most retail stores have loyalty / rewards schemes **[1] [2] [3] [4]**, it would be a severe disadvantage if retail store customers couldn't access these reward schemes when using our SCAN-N-GO app. We need access to the retail store's loyalty program database to get information on point allocation for each product and access to update user's points. We would also need access to reward schemes for rewarding multiple coffee purchases or carwash purchases.

To overcome this technical challenge, we need to have access to the store's point of sales APIs. During the onboarding process, the retail store would grant us access to the APIs. We understand that some retail stores franchises have their own custom built point of sales systems. We would have to collaboratively work with the store franchise's IT department to build connections to their point sales.

However, not all retail stores have custom built solutions, to help us overcome this technical challenge, we could also focus on implementing our solution around commonly used point of sale systems such as SAGE, SQUARE, QUICKBOOKS **[5]**. This would enable quicker onboarding for retail stores that use the above systems.

Technical challenge of Preventing Store Theft

SCAN-N-GO understands the concerns around using our app which could potentially enable shoplifting in store whether by accidental shoplifting or purposeful shoplifting. While we have added multiple security measures to our prototype, such as an Admin security dashboard, screenshot-proof digital receipt and a store scan-in system, we know there are other measures that we could put in place to help prevent shoplifting but are currently a technical challenge **[Appendix 1]**, these include:

On user registration, require user upload of a valid form of identification and a recent photo of the user's face, to be verified either in a manual process or an AI automated process. This will help security staff to identify customers in store when necessary.

Another feature is displaying customers' location in store on the Admin dashboard. For larger stores this would provide greater visibility of users and help staff monitor areas of high customer footfall or value items. We would need the user's mobile phone location to display the user's current location.

Technical challenge of Implementing Security standards
Since dealing with user data and integrating with retail store APIs, We would like to implement industry security standards such as two factor authentication for the Admin login **[6]**.

In this remit, we also need to implement proactive GDPR measures **[7]** when dealing with Retail store data and our user data.

7. Source Code Highlights

During the development of SCAN-N-GO we had encountered problems that we needed to develop smart solutions to. Below, we have included four instances in which we thought the code written was very effective. These code highlights include code from both the frontend and backend.

Python Code Highlights

AddBasketItem API Request

During the initial development of SCAN-N-GO, we realised that we needed to configure our backend Django server so that when a user attempts to add an item to the basket, the server will only create a new basket item if one does not already exist for that product in their active basket. We also needed the server to be able to account for dynamic quantities being requested by the user. We built the AddBasketItemSerializer to achieve this. We have also use this same code for the RemoveBasketItem API call to dynamically *remove* items from the cart **(Code explained with comments).**

```python
class AddBasketItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = BasketItems
        fields = ['product_id']
    '''
    Add a basket item to the database
    '''
    def create(self, validated_data):
        product_id = validated_data['product_id'] # get the product id from the request body
        request = self.context.get('request', None)
        quantity = 1 # set the quantity as default 1
        if "quantity" in request.data:
          # if there is a quantity specified in the request body then set that as the quantity
          quantity = int(request.data['quantity'])
        if request:
            current_user = request.user # get the user that made the request
            shopping_basket = Basket.objects.filter(user_id=current_user, is_active=True).first() # get the users active basket
            basket_items = BasketItems.objects.filter(product_id=product_id, basket_id_id=shopping_basket).first() # get any basket items that have the same product ID and basket ID
            if basket_items: # if there is already a basket item with that product, add on *quantity* to the basket item
                basket_items.quantity = basket_items.quantity + quantity  # if it is already in the basket, add to the quantity
                basket_items.save()
                return basket_items
            else: # if there is no basket items with that product ID and basket ID then create a new basket item with the specified quantity
                new_basket_item = BasketItems.objects.create(basket_id=shopping_basket, product_id=product_id, user_id=current_user, quantity=quantity)
                new_basket_item.save()
                return new_basket_item
        else:
            return None
```

See bigger view of code snippet in **[Appendix 3]**

UserRegistration API Request

While testing the SCAN-N-GO platform via postman, we wanted to be able to force the User Registration API into registering a user with a specific User ID, rather than simply assigning the next available ID. To achieve this we created the following piece of code. We also wanted to be able to upload profile images using the API, so we changed from JSON data to form data to achieve this **(Code explained with comments).**

```python
class UserRegistrationSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = APIUser
        fields = ['first_name','last_name','username', 'email', 'password','user_image', 'id']
        extra_kwargs = {'password': {'write_only': True}}

    '''
    Create a user, if an ID is specified, force that ID for the user (as long as its available)
    '''
    def create(self, validated_data):
        request = self.context.get('request', None) # get the request data
        # get user detail from form data of request
        email = validated_data['email']
        first_name = validated_data['first_name']
        last_name = validated_data['last_name']
        user_image = request.FILES['user_image']
        username = validated_data['username']
        password = validated_data['password']
        if "id" in request.GET:
            # if there is an id provided in the request, try to create a user with that specific id, unless one exists, in which case create the user with the next available ID
            forced_user_id = request.__getitem('id') # try to get the id from the request
            if list(APIUser.objects.filter(id=int(forced_user_id))) == []:
                # if there is no user with that id already, create the user with that specific ID
                new_user = APIUser.objects.create_user(id=forced_user_id,username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
            else:
                # if there is a user with that ID, create the user with the next available ID
                new_user = APIUser.objects.create_user(username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
        else:
            new_user = APIUser.objects.create_user(username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
        new_user.save()  # Save the new user
        new_basket = Basket.objects.create(user_id=new_user)  # Create a new empty shopping basket
        new_basket.save()  # save the shopping basket
        return new_user
```

See bigger view of code snippet in **[Appendix 4]**

Javascript Code Highlights

BackendServer()

When we deployed our SCAN-N-GO technology to Microsoft Azure web services, we realised that the front end was no longer able to communicate with the backend due to changes in the URL's. To fix this issue we created a javascript function that would figure out if the platform is being used on the scanngo.ie domain or locally, and from this it is able to generate the correct backend server URL **(Code explained with comments).**

```
function backendServer() { // get the backend server URL
  const domain = window.location.hostname.toString(); // get the current sessions domain
  if (domain === "scanngo.ie") {
    // if the domain is scanngo.ie then the backend server URL will be https://www.backend.scanngo.ie/
    var backendServerURL = "https://www.backend.scanngo.ie/";
  } else if (domain === "www.scanngo.ie") {
    // if the domain is www.scanngo.ie then the backend server URL will be https://www.backend.scanngo.ie/
    var backendServerURL = "https://www.backend.scanngo.ie/";
  } else if ( (domain === "127.0.0.1") || (domain == "localhost") || (domain == "0.0.0.0") ) {
    // if the domain is localhost or similar then the backend server URL will be 127.0.0.1:8000
    var backendServerURL = "http://127.0.0.1:8000/";
    } else {
    // if the domain is none of the above, trigger an error message
    alert("ERROR: Cannot determine Backend Server (Django) URL");
    }
  return backendServerURL // return the Backend Server URL to the function calling it
}
```

See bigger view of code snippet in **[Appendix 5].**

UserChecks()

We developed several “user checks” that will redirect the user away from pages that are inappropriate for their user journey. These include checks to make sure that Admin registered users are not on customer-facing pages and vice versa, as well as checks to make sure that users that have not logged into a store cannot access pages such as the product pages, checkout etc. These functions are then triggered on the appropriate pages and contexts.

(In this case **StoreID:5** is a **“*noStore*”** object, meaning that the user is not logged in.)

```
function redirectAdmin() {
    if (sessionStorage.getItem( key: "admin") === "true") {
        location.href = '/admin';
    }
}

function redirectUser() {
    if (sessionStorage.getItem( key: "admin") === "false") {
        location.href = '/';
    }
}

function redirectStoreLoggedInUser() {
    if (sessionStorage.getItem( key: 'storeID') !== '5' && sessionStorage.getItem( key: 'storeID') !== null) {
        location.href = '/';
    }
}

function redirectNonStoreLoggedInUser() {
    if (sessionStorage.getItem( key: 'storeID') === '5') {
        location.href = '/store-scanner';
    } else if (sessionStorage.getItem( key: 'storeID') === null) {
        location.href = '/store-scanner';
    }
}
```

See bigger view of code snippet in **[Appendix 6].**

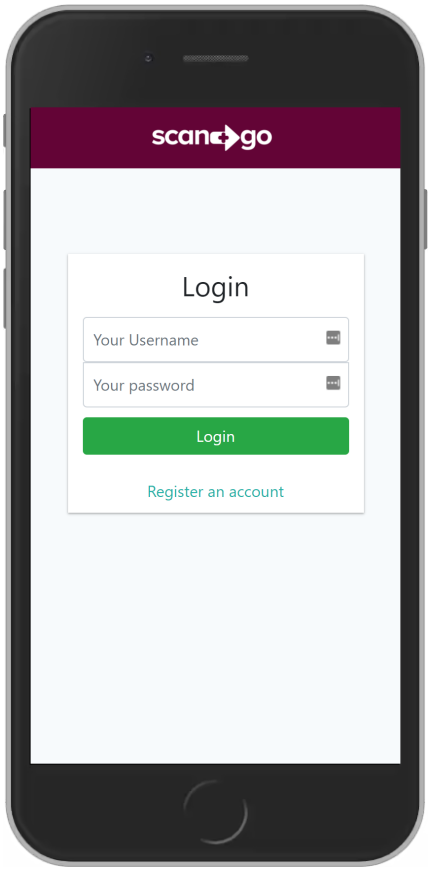
8. User Interface Justification

Mobile Web Application

SCAN-N-GO has two distinct user interfaces that each customer segment interacts with, the web-application interface and Admin interface. In both of these interfaces careful consideration for various aspects of the design were considered.

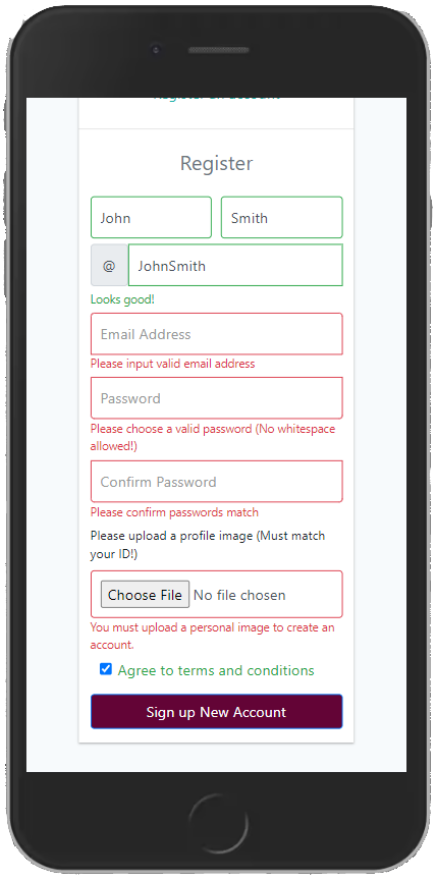
Our core design principles include;

- Clear CTAs (Consistent colouring)
- Clear Copy / Language
- Large icons / buttons
- Use of images / Minimised copy
- Provide feedback to the user



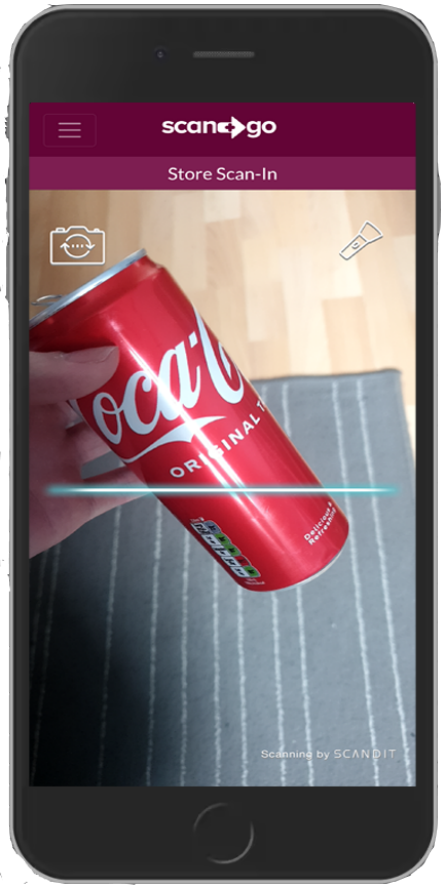
Login / Registration Page

Our login page has a large green call to action which prompts the user to "Login", this language was chosen purposefully in order to avoid potential confusion for the user. For example, "Sign In" has the potential to be misread by the user as "Sign Up". Furthering on from this, the use of white space draws attention to the login process. The option to register for an account is presented for the user which is presented to the user as a dropdown. Once the user progresses past this step they will be brought to the store scan-in page.



Login / Registration Page

The registration form is a clear demonstration of one of our design principles of providing feedback to the user. For each invalid field, a small description of the error is displayed in order to allow the user to amend the error promptly. Once the user progresses past this step they will be brought to the store scan-in page.

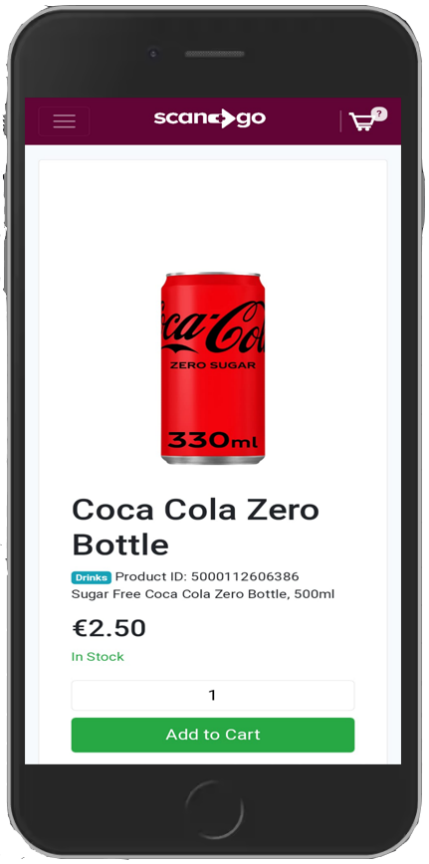


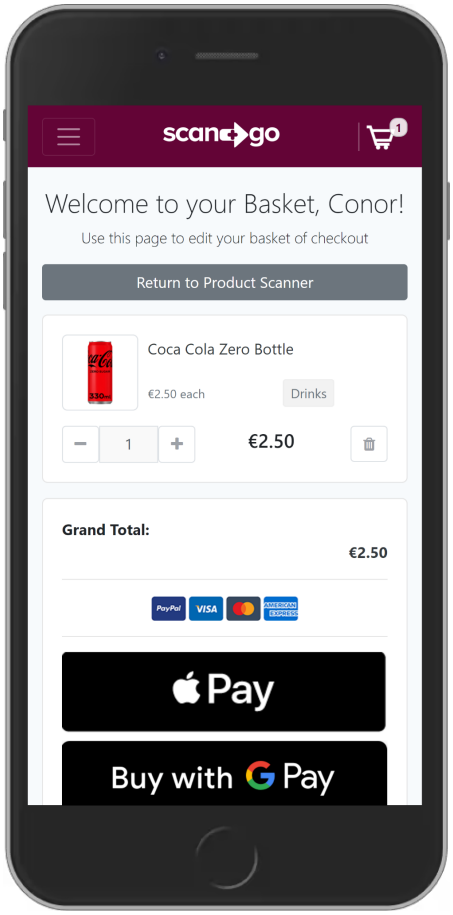
Store / Product Scan-In

The store scan-in / product page displays the user's device's camera. Two unintrusive icons are on the page which allow the user to turn on their flashlight as well as use their front-facing camera. The rear facing camera is enabled by default. This allows the user to easily scan the store scan-in tag / product. The hamburger icon located within the navigation bar allows the user to view previous orders at this point. Once they have scanned into a store, a basket icon will appear in the top right corner.

Product Page

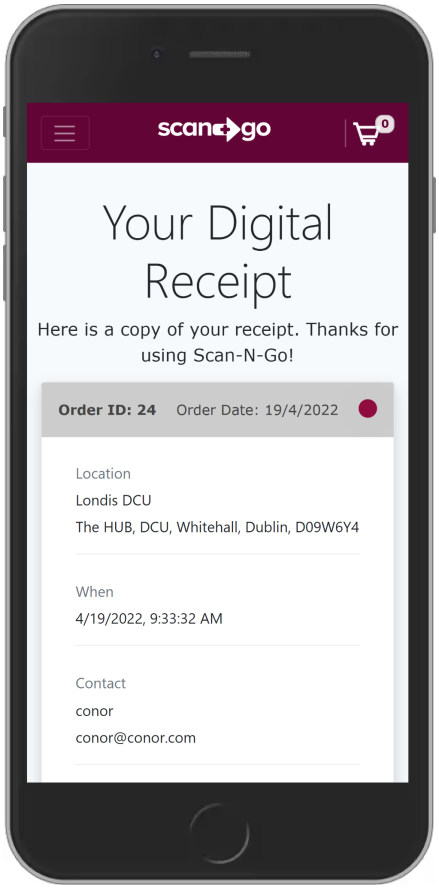
For the item product page we make use of a white background which provides contrast to the image. An intuitive item amount selection is located close to the “Add to Cart” button which is also consistent with our Call-to-action colouring scheme to ensure the correct flow is clear to the user at all times. Once a user adds an item/items to their cart they will be taken back to the product scanner page. The bubble within the cart which is in the navigation bar will indicate how many items there are in their cart at any given time.





Basket Page

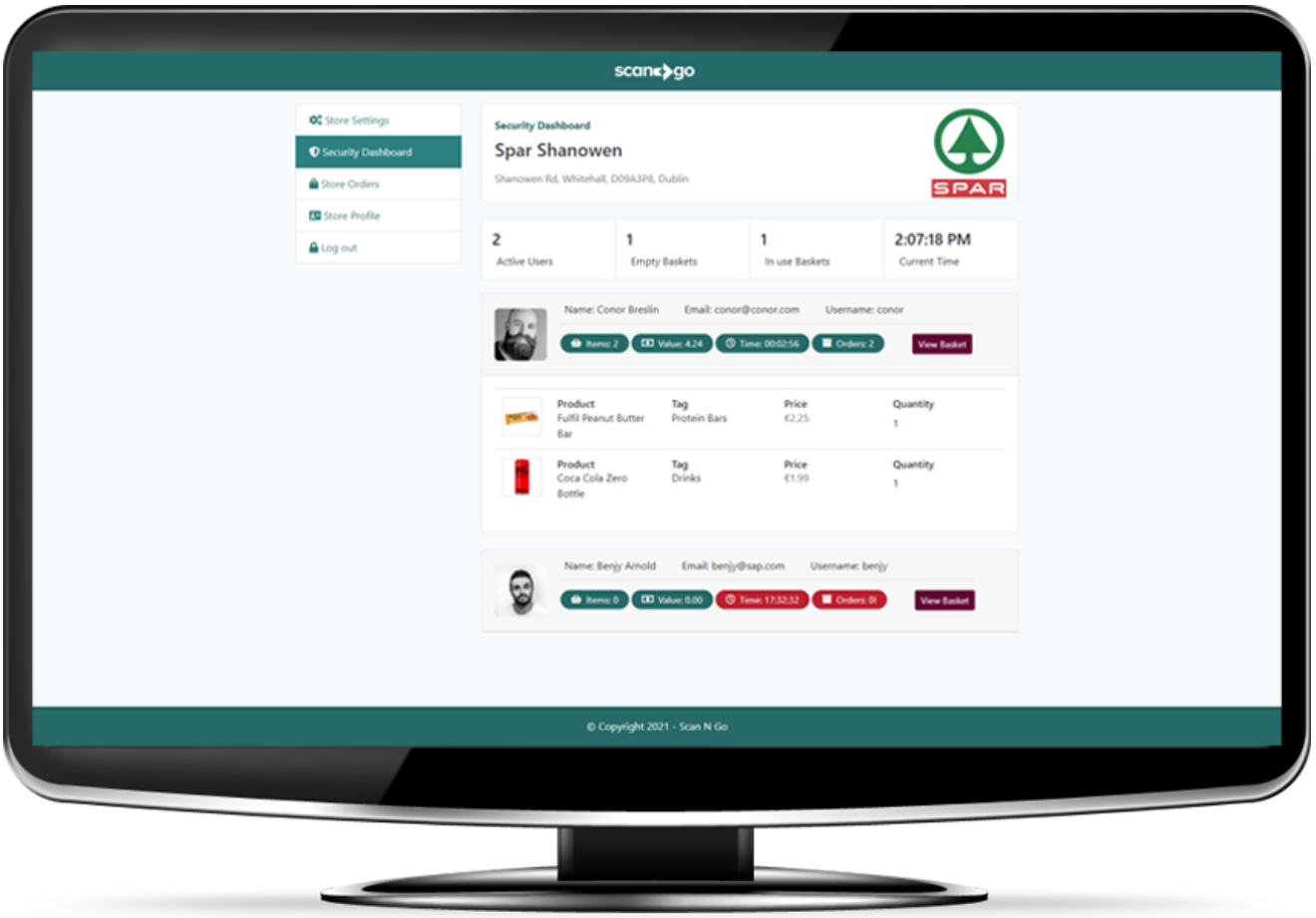
We designed this Basket / shopping cart page with a clear focus on buying the product. We reduced the number of action buttons to make the user focus on the checkout process. The user’s limited action buttons are the payment options (google pay, apple pay etc), the trashcan to remove a product and the exit (back to scanner) button. We use the white product tiles to reduce distractions to our uses, and we emphasise the payment buttons by making them the largest visible button on the viewport.



Basket Page

We designed the digital receipt page with the core intention of allowing users to view all previous orders on their account. As this may amount to many orders over time, all receipts are placed within a dropdown accordion, which reduces the amount of scrolling necessary. Our digital receipt page is also dynamic in the way that the latest receipt has an icon that will pulsate. This is a security feature. All information regarding each purchase is available to the user within the dropdowns. The next steps for the users at this point are to either logout or return to the product scanner to scan more products.

Admin Web Application



Security Dashboard

The purpose of our security dashboard is to display information of the users that are currently in-store. A design choice that was made based on user feedback was to alter the colouring from the customer web-app - hence, the differing colour scheme. Information is legible at a glance making efficient use of whitespace as well as accordion style dropdowns. Quick-view information bubbles are displayed with variable colouring based on different variables (Long time in-store, no items in basket, etc). This design is echoed across all aspects of the admin dashboard.

Appendix 1

Document detailing the security risks associated with Scan-N-Go and techniques for mitigating these risks.

Security

SCAN-N-GO

Name: Jamie Behan
Student Number: 18406986
Email: Jamie.Behan6@mail.dcu.ie

Name: Bernard McWeeney
Student Number: 18384466
Email: Bernard.McWeeney2@mail.dcu.ie

Name: Shaun Kee
Student Number: 18308546
Email: Shaun.Kee2@mail.dcu.ie

Module Code: CA472
Date of Submission: 19/04/2022
Programme: EC4

1

Security Issues

- **Shoplifting**
- **Users purchasing restricted items (Alcohol, Cigarettes, etc,)**

Current Security Measures in Retail Industry

Self service tills current security implementations

- Weights at till
- Staff Awareness
- Lights on self service stills if error occurs
- Security before and after Self Service

Human operated tills current security implementations

- Staff Awareness
- Shop Cameras
- Security before and after store entry
- Legitimate Cash checker (Pens, scanner)

Security Measures

- **Digital Receipt**
  - When payment is completed, the customer is provided with a digital receipt of their purchase.
  - Timer set when the purchase is made, to show how recent the receipt is and prevent fraudulent receipts.

- **Admin dashboard**
  - Provides visibility to staff (Displays active users in-store)
  - Displays each customer's current baskets and items
  - Displays User store ratings based on previous orders - gives guidance to staff on legitimacy of users.
  - Profile picture
  - Order history

- **Noise when scanning a product and scanning into a store**
  - Provides feedback to users of successful scan
  - Alerts staff to active users in store

- **Creation of store culture of being watched. - rephrase - reminder of monitoring in process**
  - In-app prompts / messages
  - In-store posters and signage

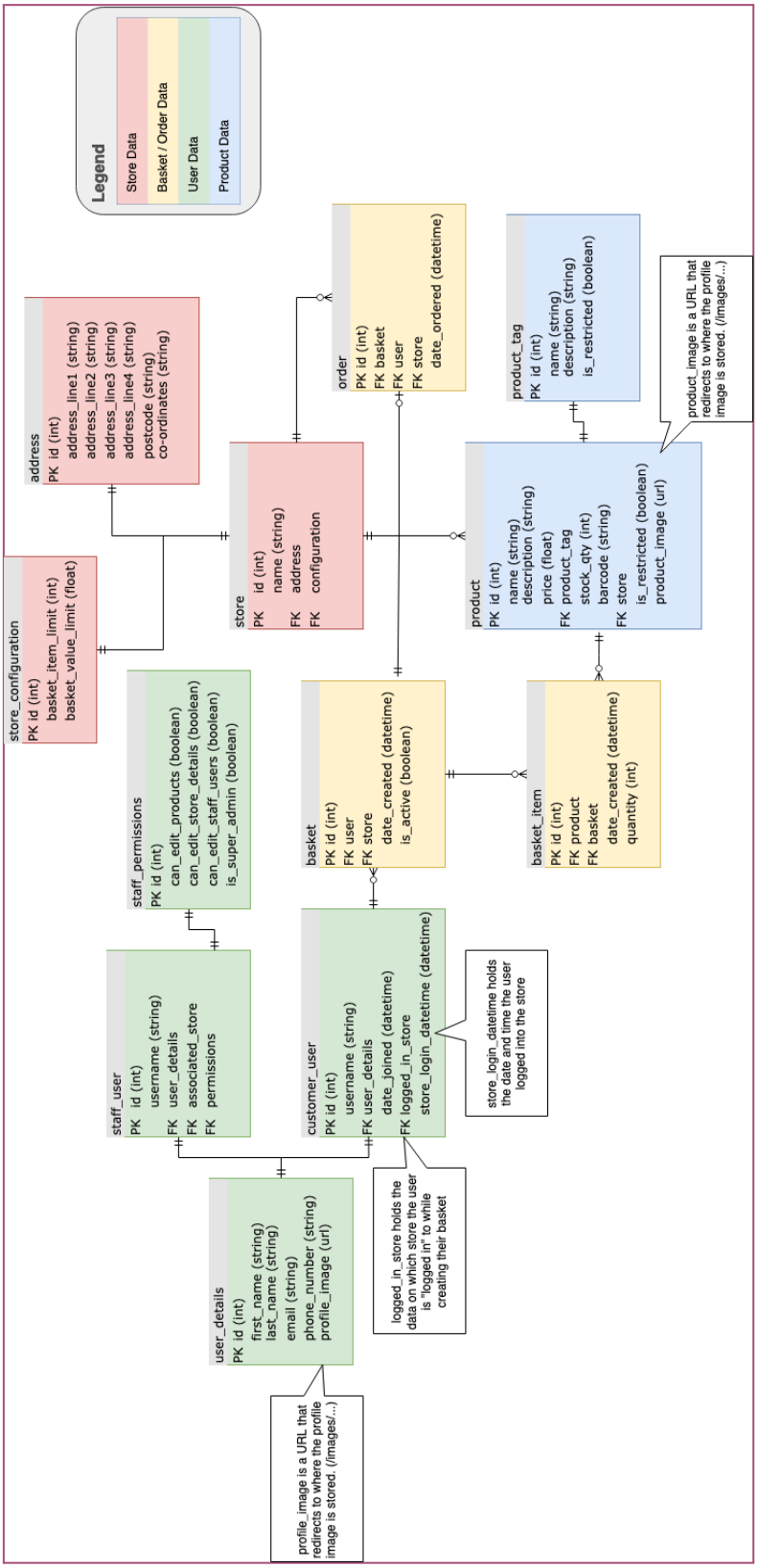
- **Staff training**
  - Upskill staff members to understand Scan-N-Go technology

- **Restrictive items require staff approval**
  - QR Code displays waiting for Staff Approval

- **Option for store to limit Quantity of Items**
  - Limits the users to a certain amount of items

- **Option for store to limit total basket value**
  - Limits the users to a certain amount of items

Appendix 2


Legend
Store Data
Basket / Order Data
User Data
Product Data
address
PK id (int)
address_line1 (string)
address_line2 (string)
address_line3 (string)
address_line4 (string)
postcode (string)
co-ordinates (string)
order
PK id (int)
FK basket
FK user
FK store
date_ordered (datetime)
product_tag
PK id (int)
name (string)
description (string)
is_restricted (boolean)
product_image is a URL that redirects to where the profile image is stored. (/images/...)
store_configuration
PK id (int)
basket_item_limit (int)
basket_value_limit (float)
store
PK id (int)
name (string)
FK address
FK configuration
product
PK id (int)
name (string)
description (string)
price (float)
FK product_tag
stock_qty (int)
barcode (string)
FK store
is_restricted (boolean)
product_image (url)
staff_permissions
PK id (int)
can_edit_products (boolean)
can_edit_store_details (boolean)
can_edit_staff_users (boolean)
is_super_admin (boolean)
basket
PK id (int)
FK user
FK store
date_created (datetime)
is_active (boolean)
basket_item
PK id (int)
FK product
FK basket
date_created (datetime)
quantity (int)
staff_user
PK id (int)
username (string)
FK user_details
FK associated_store
FK permissions
customer_user
PK id (int)
username (string)
FK user_details
date_joined (datetime)
FK logged_in_store
store_login_datetime (datetime)
store_login_datetime holds the date and time the user logged into the store
user_details
PK id (int)
first_name (string)
last_name (string)
email (string)
phone_number (string)
profile_image (url)
logged_in_store holds the data on which store the user is "logged in" to while creating their basket
profile_image is a URL that redirects to where the profile image is stored. (/images/...)

Appendix 3

```python
class AddBasketItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = BasketItems
        fields = ['product_id']
    '''
    Add a basket item to the database
    '''
    def create(self, validated_data):
        product_id = validated_data['product_id'] # get the product id from the request body
        request = self.context.get('request', None)
        quantity = 1 # set the quantity as default 1
        if "quantity" in request.data:
            # if there is a quantity specified in the request body then set that as the quantity
            quantity = int(request.data['quantity'])
        if request:
            current_user = request.user # get the user that made the request
            shopping_basket = Basket.objects.filter(user_id=current_user, is_active=True).first() # get the users active basket
            basket_items = BasketItems.objects.filter(product_id=product_id, basket_id_id=shopping_basket).first() # get any basket items that have the same product ID and basket ID
            if basket_items: # if there is already a basket item with that product, add on *quantity* to the basket item
                basket_items.quantity = basket_items.quantity + quantity  # if it is already in the basket, add to the quantity
                basket_items.save()
                return basket_items
            else: # if there is no basket items with that product ID and basket ID then create a new basket item with the specified quantity
                new_basket_item = BasketItems.objects.create(basket_id=shopping_basket, product_id=product_id, user_id=current_user, quantity=quantity)
                new_basket_item.save()
                return new_basket_item
        else:
            return None
```

Appendix 4

```
class UserRegistrationSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = APIUser
        fields = ['first_name','last_name','username', 'email', 'password','user_image', 'id']
        extra_kwargs = {'password': {'write_only': True}}

    '''
    Create a user, if an ID is specified, force that ID for the user (as long as its available)
    '''
    def create(self, validated_data):
        request = self.context.get('request', None) # get the request data
        # get user detail from form data of request
        email = validated_data['email']
        first_name = validated_data['first_name']
        last_name = validated_data['last_name']
        user_image = request.FILES['user_image']
        username = validated_data['username']
        password = validated_data['password']
        if "id" in request.GET:
            # if there is an id provided in the request, try to create a user with that specific id, unless one exists, in which case create the user with the next available ID
            forced_user_id = request.__getitem('id') # try to get the id from the request
            if list(APIUser.objects.filter(id=int(forced_user_id))) == []:
                # if there is no user with that id already, create the user with that specific ID
                new_user = APIUser.objects.create_user(id=forced_user_id,username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
            else:
                # if there is a user with that ID, create the user with the next available ID
                new_user = APIUser.objects.create_user(username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
        else:
            new_user = APIUser.objects.create_user(username=username,first_name=first_name,last_name=last_name,email=email, password=password,user_image=user_image)
        new_user.save()  # Save the new user
        new_basket = Basket.objects.create(user_id=new_user)  # Create a new empty shopping basket
        new_basket.save()  # save the shopping basket
        return new_user
```

Appendix 5

```
function backendServer() { // get the backend server URL
  const domain = window.location.hostname.toString(); // get the current sessions domain
  if (domain === "scanngo.ie") {
    // if the domain is scanngo.ie then the backend server URL will be https://www.backend.scanngo.ie/
    var backendServerURL = "https://www.backend.scanngo.ie/";
  } else if (domain === "www.scanngo.ie") {
    // if the domain is www.scanngo.ie then the backend server URL will be https://www.backend.scanngo.ie/
    var backendServerURL = "https://www.backend.scanngo.ie/";
  } else if ( (domain === "127.0.0.1") || (domain == "localhost") || (domain == "0.0.0.0") ) {
    // if the domain is localhost or similar then the backend server URL will be 127.0.0.1:8000
    var backendServerURL = "http://127.0.0.1:8000/";
    } else {
    // if the domain is none of the above, trigger an error message
    alert("ERROR: Cannot determine Backend Server (Django) URL");
    }
  return backendServerURL // return the Backend Server URL to the function calling it
}
```

Appendix 6

```
function redirectAdmin() {
    if (sessionStorage.getItem( key: "admin") === "true") {
        location.href = '/admin';
    }
}

function redirectUser() {
    if (sessionStorage.getItem( key: "admin") === "false") {
        location.href = '/';
    }
}

function redirectStoreLoggedInUser() {
    if (sessionStorage.getItem( key: 'storeID') !== '5' && sessionStorage.getItem( key: 'storeID') !== null) {
        location.href = '/';
    }
}

function redirectNonStoreLoggedInUser() {
    if (sessionStorage.getItem( key: 'storeID') === '5') {
        location.href = '/store-scanner';
    } else if (sessionStorage.getItem( key: 'storeID') === null) {
        location.href = '/store-scanner';
    }
}
```

References

1. Tesco Clubcard, [WWW Document], URL: https://www.tesco.ie/clubcard/
2. Customer Rewards Card, Coupons & More - Lidl Plus App, [WWW Document], URL: https://www.lidl.ie/lidl-plus
3. Rewards | Londis, [WWW Document], URL: https://www.londis.ie/rewards/
4. Rewards - Spar, [WWW Document], URL: https://www.spar.ie/rewards/
5. Top 9 Best Point of Sales (POS) Systems – 2022 Review, [WWW Document], URL: https://www.quicksprout.com/best-point-of-sales-systems-pos/
6. What's multi-factor authentication - and why should I care?, [WWW Document], URL: https://www.nist.gov/blogs/cybersecurity-insights/back-basics-whats-multi-factor-authentication-and-why-should-i-care
7. What is GDPR, the EU's new data protection law?, [WWW Document], URL: https://gdpr.eu/what-is-gdpr/
8. How to Make a Data Flow Diagram, [WWW Document], URL: https://www.lucidchart.com/pages/data-flow-diagram/how-to-make-a-dfd