



Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

DigiWallet – Final Report

Emily Gavin

G00382828

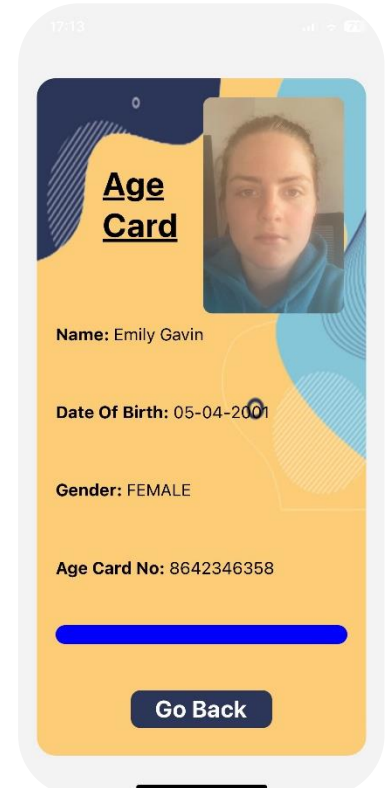
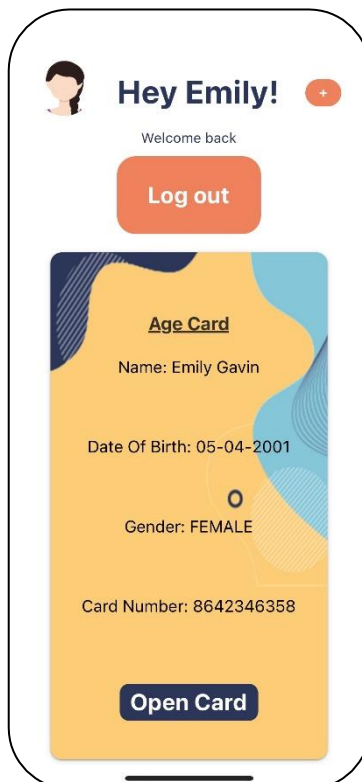
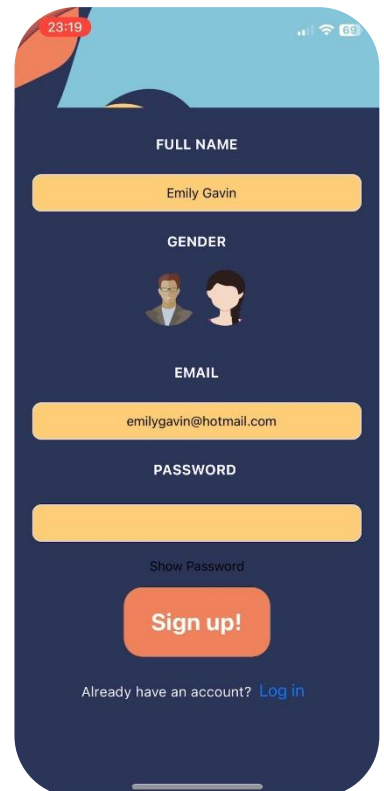
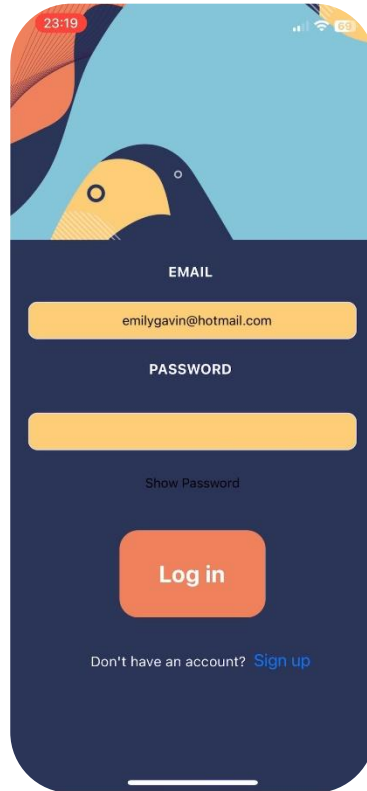
BEng(H) in Software & Electronic Engineering

Project Engineering

Atlantic Technological University Galway

2022/2023

Preview



Declaration

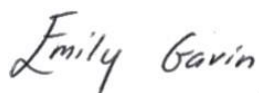
This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University Galway.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

ChatGPT was used at various points of this report to enhance the clarity and overall quality of the writing. This was used to refine some paragraphs, making it more clear and engaging for the readers.

This allowed the report to effectively communicate the purpose, design, and implementation of the DigiWallet project, ensuring that readers could easily understand the project's goals, technologies used, and the challenges faced during its development.

Signed,

A handwritten signature in black ink that reads "Emily Gavin". The script is cursive and fluid, with the first letters of each name being capitalized and prominent.

Acknowledgements

I express my thanks to my beloved family and friends for the unwavering support they have shown me throughout the years. I am also grateful to the academic staff of ATU for their invaluable guidance and support during my studies.

In particular, I extend my appreciation to Brian O'Shea for his support and encouragement throughout my project.

I would also like to acknowledge my fellow classmates, who have become a tight-knit group over the past few years. I wish every one of you the very best in your future endeavours.

Table of Contents

1	Summary.....	8
2	Poster	9
3	Introduction	10
4	Background Research.....	11
5	Project Architecture Diagram	12
6	Project Plan.....	13
7	Back End Set-up	14
7.a	Spring Initializr	14
7.1	Spring Initializr Set-up.....	14
7.b	JSON Mock-up.....	15
7.2	JSON Mock-up.....	15
7.c	User Class.....	16
7.3	User Class Examples.....	16
7.d	Connecting to Docker.....	17
7.4	docker-compose file.....	17
7.e	Application Properties.....	18
7.5	application.properties.....	18
8	Back End Implementation.....	19
8.a	Spring Boot Architecture.....	19
8.1	Spring Architecture Diagram.....	29

8.b	Controller Class.....	20
	8.2 Controller Examples.....	20
8.c	Service Class.....	21
	8.3 Service Examples.....	21
8.d	Repository Interface.....	22
	8.4 Repository Example.....	22
8.e	Password Encryption and Decryption.....	23
	8.5 Encryption Function.....	24
	8.6 Decryption Function.....	24
9	Front End Set-up.....	25
	9.a Installation.....	25
	9.1 Expo and Ngrok.....	25
10	Front End Implementation.....	26
	10.a Native Stack Navigator.....	26
	10.1 Native Stack Navigator.....	26
	10.b Folder Organisation.....	27
	10.2 Folder Organisation.....	27
	10.c User Screens.....	28
	10.3 Log In Screen.....	28
	10.4 Sign Up Screen.....	29
	10.5 Welcome Screen.....	29
	10.d Main Screen.....	30

10.6 Add New Card Screen.....	31
10.e Card Screen.....	32
10.7 Card Screen.....	32
10.f Main Screen Progression Example.....	33
10.8 Main Screen Cycle.....	33
11 Cloud Connection.....	34
11.a Mongo Atlas.....	34
11.1 Mongo Atlas Cluster.....	34
11.2 Postman Request.....	34
11.b AWS EC2 Instance.....	35
11.3 EC2 Instance Connect.....	36
12 Testing.....	37
12.a Unit Tesing.....	37
12.1 Unit Testing Example.....	37
13 Future Development.....	38
14 Conclusion.....	39
15 References.....	40

1. Summary

DigiWallet is a digital identification app that allows users to store their Identification cards in one place. My app is developed using React Native for the frontend, which is hosted on AWS, and a Java Spring Boot backend project connected to a Docker container containing a MongoDB database connected to Mongo Atlas.

There were two main driving forces behind the creation of DigiWallet. Firstly, in light of the recent pandemic, there has been a huge shift towards digital payment methods, indicating that our society is heading towards a cashless future. I believe that the way we prove our identity should also adapt to this trend. Instead of relying on physical cards, there should be an alternative way to verify our age, address, and identity in a digital form.

Secondly, we strive to contribute to a more sustainable environment by reducing our plastic consumption. Traditional plastic ID cards are manufactured from non-renewable PVC materials, which can cause significant environmental damage. Moreover, due to their durability, these cards can take a long time to break down after disposal. Using technology-based photo ID cards is a step towards reducing our environmental impact and promoting sustainable practices.

The frontend of the app is developed using React Native, a popular framework for building mobile apps that can be run on both iOS and Android platforms. The app is hosted on AWS, a cloud computing service that provides a reliable and scalable infrastructure for hosting web applications.

To use the app, users can simply upload their identification cards by taking a photo of them using the camera on their mobile device. The user is then asked to enter the relevant information from their cards such as name, date of birth, address, card number etc. to build an app interface. This information is then securely stored in the MongoDB database.

Overall, DigiWallet provides a convenient and secure way for users to store and access their important identification documents in one place. With the use of advanced technologies such as Spring Boot, Docker, MongoDB, React Native, and AWS, the app is able to provide a seamless user experience.

2. Poster

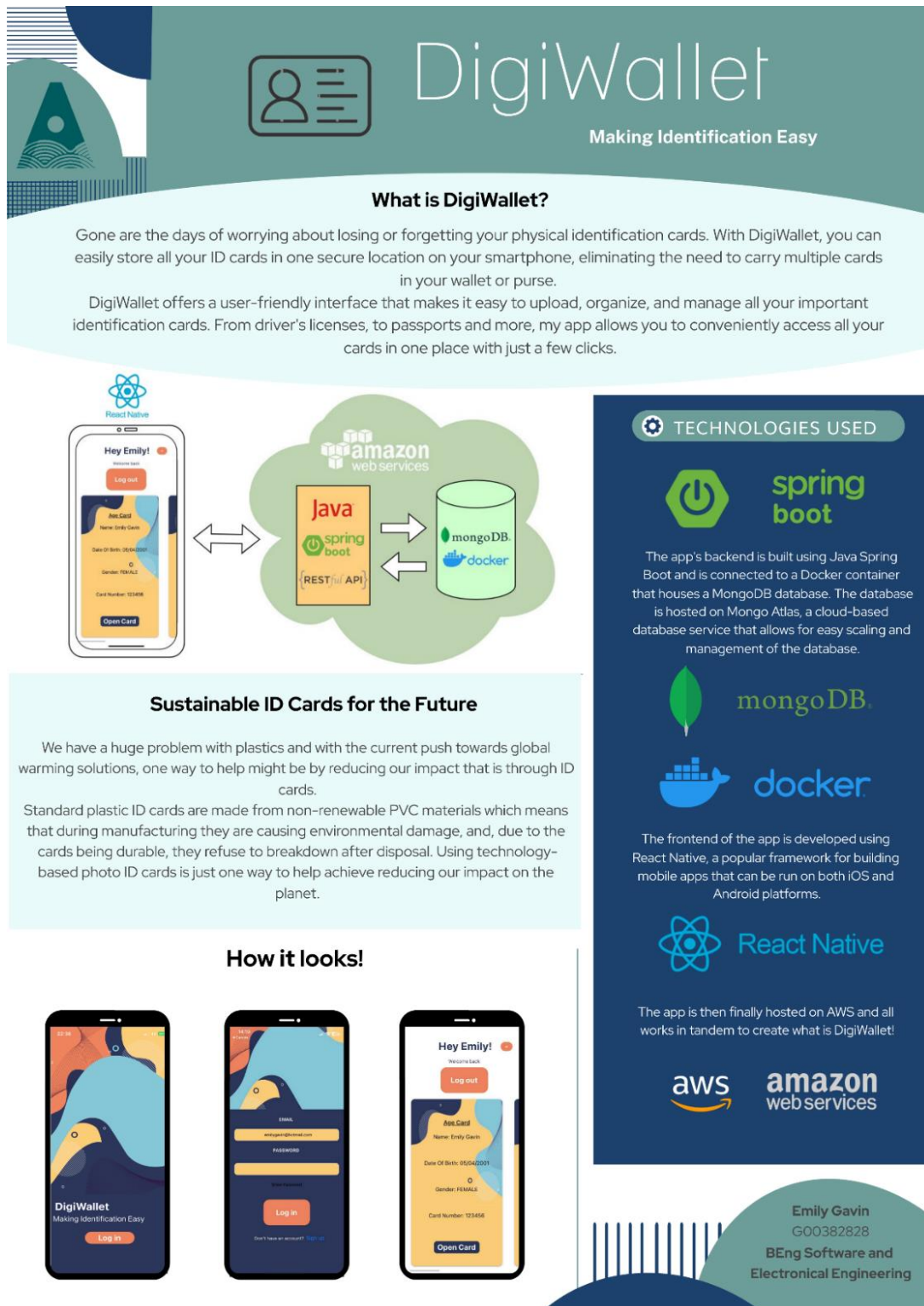


Figure 2.1 Poster

3. Introduction

The DigiWallet project utilizes several modern technologies to create a seamless and secure user experience. The back end of the app is built using Java Spring Boot which is also connected to a Docker container containing a MongoDB database, which is hosted on Mongo Atlas and AWS.

Finally, the app's frontend is created using React Native and connects to my phone using an Expo tunnel.

The goal of this project is to provide a secure and convenient solution for users to manage their personal identification cards. By storing their identification documents digitally, users can reduce the risk of losing or damaging their physical cards and have access to them at any time from their mobile device. I set out the goal for the app to be able to save and display a user's Student Card, Drivers License, Age Card, and Passport Card all in one place. I wanted to create an intuitive and aesthetically pleasing application interface that worked in tandem with my backend to successfully create DigiWallet. Overall, the project aims to provide a seamless and efficient user experience that makes managing personal identification documents easier and more convenient.

In this report, I will provide a detailed overview of the DigiWallet project, including its architecture, design, and implementation. I will also discuss the challenges I faced during the project and how I overcame them. Finally, I will conclude with an evaluation of the project's success and the lessons I learned from this experience.

4. Background Research

Before starting DigiWallet, there were several key areas of background research that had to be undertaken to ensure my projects success. Here are some areas I considered:

Spring Boot: As the backend of the DigiWallet project is developed using Java and Spring Boot, it was important to have a solid understanding it's technology. This includes researching its capabilities, best practices, and any potential challenges or limitations that may arise during development. I had prior experience working with Spring/Spring Boot due to my internship and was quite confident in my ability to create a functional Spring Boot application.

Docker: Docker is used to host the MongoDB database in a container. Therefore, it was essential to research how Docker works, how to set up a Docker container, and how to manage it. This was an area I also had experience in prior to the project which was then further deepened in Paul's CICD class where we learned about Docker and its capabilities.

MongoDB and Mongo Atlas: To ensure that the MongoDB database is configured correctly, research had to be done to understand the features, benefits, and limitations of MongoDB and Mongo Atlas.

React Native: The frontend of the DigiWallet project is developed using React Native, so it's crucial to understand its features, limitations, and best practices. This also included figuring out a connection method between my front end and my back end, to which I used an Ngrok tunnel which was taught to us in Brian's Mobile Application Development module.

AWS: As the application is hosted on AWS, it's important to research how to set up and manage AWS instances, how to peer to a Mongo Atlas database, and how to handle data storage and security.

5. Project Architecture Diagram

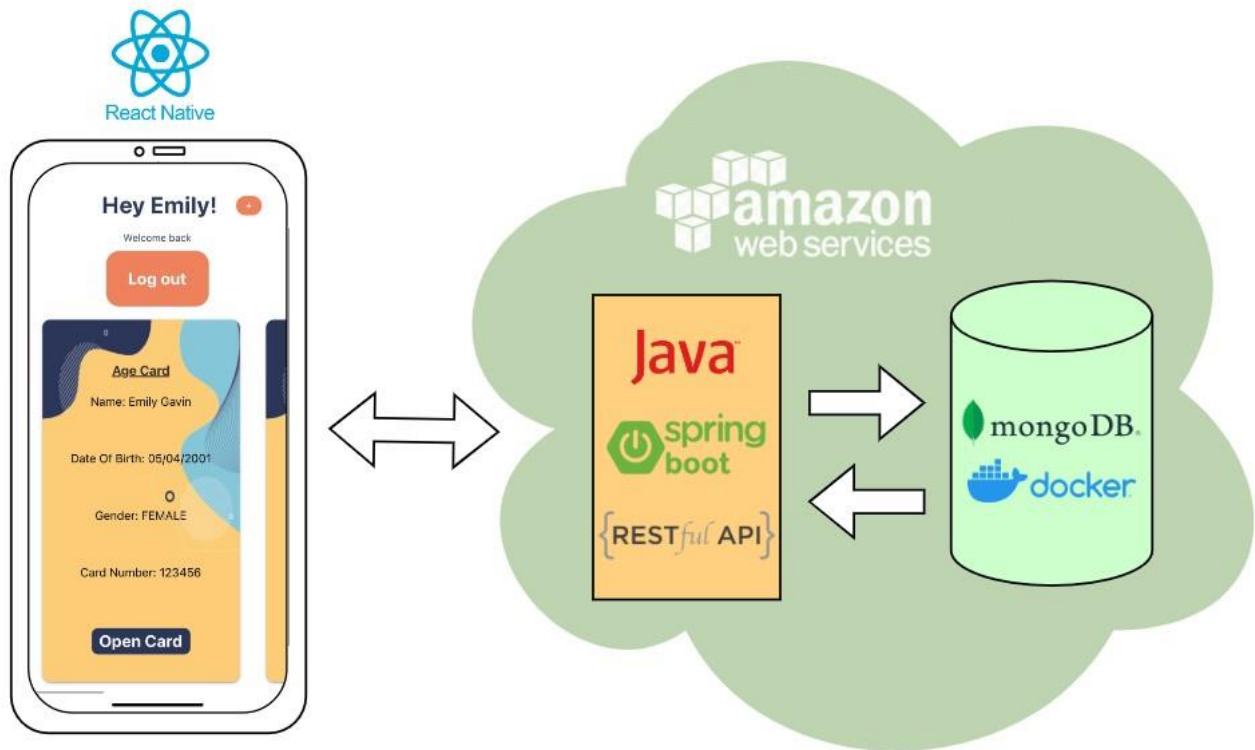


Figure 5.1 Architecture Diagram

The backend Spring Boot application for DigiWallet serves as the main processing unit for the project. It is the backbone of my application and ensures that all data is collected and given correctly. It receives requests from the frontend application and communicates with the MongoDB database to process the interactions.

The MongoDB database is hosted on a Docker container and is connected to Mongo Atlas for data storage and management. The container is used to provide a standardized and isolated environment for the database, ensuring that the application remains stable.

The frontend of the application is developed using React Native. The React Native framework allows the frontend application to be developed for both iOS and Android platforms. The frontend application communicates with the backend Java Spring Boot application through RESTful API endpoints, enabling data exchange between the two applications.

6. Project Plan

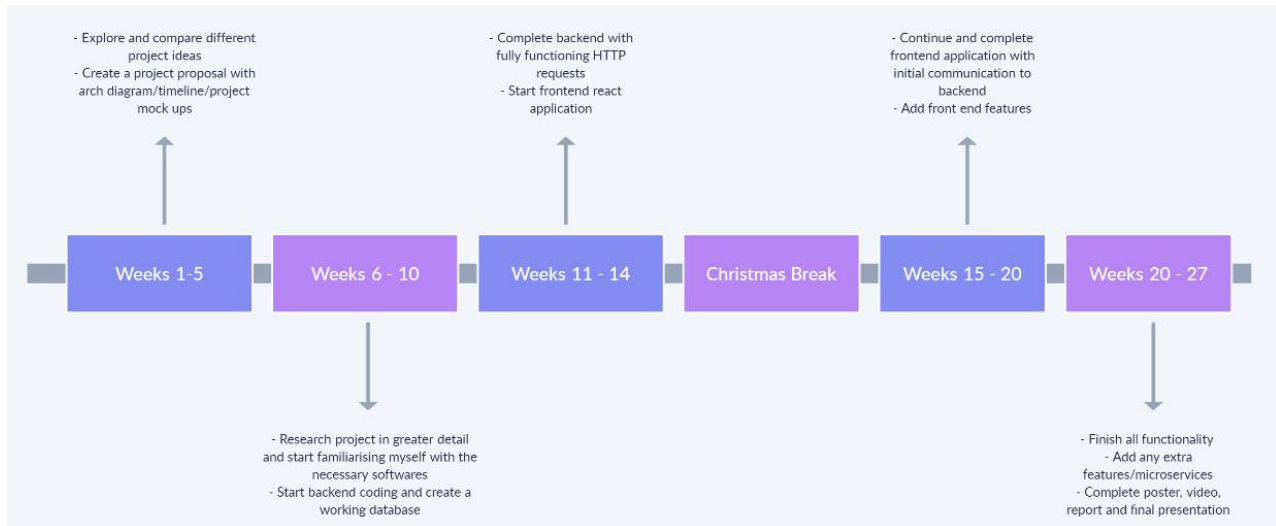


Figure 6.1 Architecture Diagram

When starting my project this was my initial Project Timeline, which ended up being very accurate to my completed timeline. My plan was as follows.

1. Research.

- I wanted to ensure I had extensive research done before getting started on my project. I wanted to do this to get a general and realistic plan put in place to follow for the year. This included comparing and choosing different software tools, an approximation on time needed for each project step and more.

2. Start building the back end using Spring Boot and connect to MongoDB and Docker.

- I wanted to start at this point as I was confident with my abilities in back-end implementation. I had previous experience with Spring/Spring Boot after my internship with Genesys and knew that this was where I was most confident to get my project started. I also had experience with Mongo DB and Docker and wanted to showcase my abilities to use these tools through my project.

3. Have all functioning HTTP endpoints (using Postman).

- I wanted to ensure that by Christmas I had an application that could send and receive data from a MongoDB database that contained my desired JSON objects. This includes having all correctly functioning GET, POST and DELETE endpoints.

4. Start front end implementation into my project.

- I knew when starting my project that front-end design would be the biggest challenge for me as I had very limited experience doing it. This is why I left this step to complete for my whole second semester, to hopefully give myself enough time to grasp enough concept to be able to create a successful app.

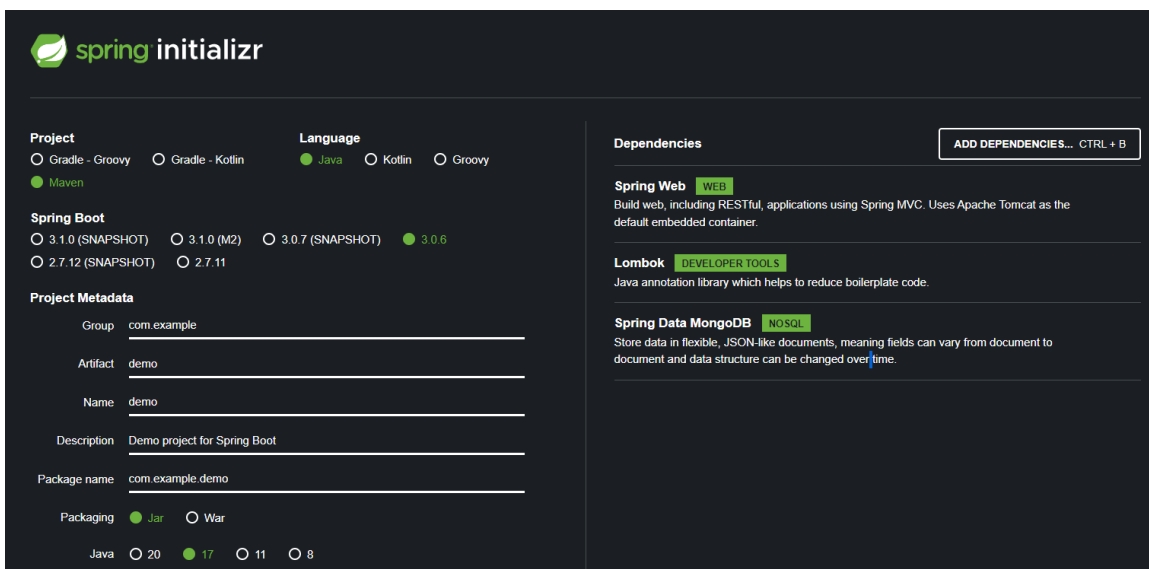
5. Connect to the cloud.

- Finally, my last step was to deploy my project to be available to use from a cloud. I felt this was an important step to finish my project and to do this I would use Mongo Atlas and AWS.

7. Back End Set-up

7.a Spring Initializr

The back-end functionality for my application has been created via Spring Boot. Java Spring Boot is an open-source tool that makes it easier to use Java-based frameworks to create microservices and web apps [1]. To start my apps creation, I used Spring Initializr. With the help of Initializr, I could easily generate the structure of my Spring Boot Project. It offers extensible API for creating JVM-based projects. Here is what my Spring Initializr set-up page looked like:



The screenshot shows the Spring Initializr web interface with the following configuration options:

- Project:**
 - ☐ Gradle - Groovy
 - ☐ Gradle - Kotlin
 - ☒ Maven
- Language:**
 - ☒ Java
 - ☐ Kotlin
 - ☐ Groovy
- Spring Boot:**
 - ☐ 3.1.0 (SNAPSHOT)
 - ☐ 3.1.0 (M2)
 - ☐ 3.0.7 (SNAPSHOT)
 - ☒ 3.0.6
 - ☐ 2.7.12 (SNAPSHOT)
 - ☐ 2.7.11
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
- Packaging:**
 - ☒ Jar
 - ☐ War
- Java:**
 - ☐ 20
 - ☒ 17
 - ☐ 11
 - ☐ 8
- Dependencies:**
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
 - Spring Data MongoDB** (NOSQL): Store data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

Figure 7.1 Spring Initializr Set-up

Spring Initializr provides a user interface where developers can choose the project's build system (Maven or Gradle), programming language (Java or Kotlin), Spring Boot version, and various dependencies such as database drivers, web frameworks, and security modules. Once the desired options have been selected, Spring Initializr generates a project structure that can be downloaded and imported into an IDE such as Eclipse or IntelliJ IDEA. In my project, I used three main dependencies.

Spring Web: A module of the Spring Framework for building web applications.

Lombok: A Java library for reducing boilerplate code in classes using annotations.

Spring Data MongoDB: A module of the Spring Framework for working with MongoDB databases, providing an easy-to-use API for CRUD operations, querying, and mapping.

7.b JSON Mock-up

Then when I downloaded my Spring Boot starter pack, I had the first look at what was to become the back-end driver for my DigiWallet project. I used IntelliJ as my IDE of choice to develop upon the generated folder. My first order of business was to create a mock JSON as to what I wanted my database structure to look like. To do this, I had to think about what fields I would need to create a log-in based application that could store ID card information. Here is a look into my first JSON mock up I created back in October:

Here, you can take a look at the general JSON idea I had for the objects that would be within my MongoDB database.

A unique ID to identify the user within the database.

The users email and password so sign up and log in capabilities would be possible.

An embedded class called cards that could hold each card object within.

Here I have an example as to how two cards, the student card, and the age card, would be stored within a user's object.

In this example, I only used two cards within the object to get an understanding of how it might look.

To achieve this embedded JSON look, I had to use embedded classes. The final object had three layers to the JSON:

First the main user information, then the cards class to hold the cards, then finally each individual card class to hold the necessary information for each ID card.

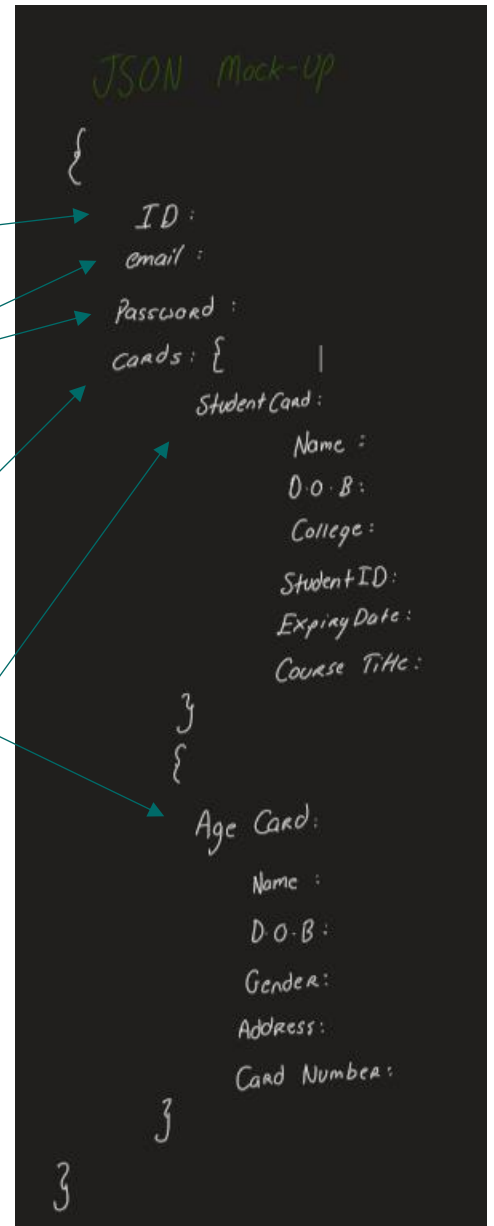


Figure 7.2 JSON Mock-up

7.c User Class

Using the JSON Mock-up, I could then create my “User” class which would be the schema that would inevitably form my objects within my database.

```
public class User {
    private String id;
    1 usage
    private String name;
    1 usage
    private String gender;
    1 usage
    @Column(unique = true)
    private String email;
    1 usage
    private String password;
    private String uri;
    1 usage
    private Cards cards;
```

This is what my final User Schema looked like which consisted of all the variable types and names.

You can see I have used the annotation `@Column(unique = true)` which is used to ensure that my email column is unique and that there are no two users with the same email existing within my database.

I also have created my own class called “Cards” which is embedded within the User class.

```
public class Cards{
    private DriversLicense driversLicense;
    private AgeCard ageCard;
    private PassportCard passportCard;
    private StudentCard studentCard;
```

My Cards class then holds four more embedded classes of the types of each card that I wanted the user to be able to hold within their account.

```
public class StudentCard{
    private String type;
    private String name;
    private String dateOfBirth;
    private String college;
    private String studentID;
    private String courseTitle;
    private String expiryDate;
}
```

Here is an example of my StudentCard class. I created each card with their own unique variables that are linked to the necessary information that I needed for each card type. As you can see, a Student Card would need information such as course title or student ID, whereas a drivers License would not need this information.

Figures 7.3 User Class Examples

7.d Connecting to Docker

For my project I used Docker to host my MongoDB database. Docker is a powerful development platform that enables users to containerize software. These containers can be run on any machine, as well as in a public or private cloud [2]. There were 3 main reasons as to why I decided to do this:

Isolation: By running MongoDB in a Docker container, I could isolate it from the host system and other containers running on the same system. This can help to ensure that my database was running in a **consistent** and **predictable** environment.

Portability: Docker containers are designed to be easily portable across different systems and environments. This makes it easier to move your database between development, testing, and production environments, without having to worry about differences in the underlying infrastructure.

Security: By running MongoDB in a Docker container, I could implement additional security measures to protect my database. This was especially important for me given that security was something I was adamant about since day one, and knowing this, I knew I had to use Docker.

I used a docker-compose.yml file to initialise my docker containers.

```
services:
  digiwallet:
    image: mongo
    container_name: digiwallet
    ports:
      - 27017:27017
    volumes:
      - mydata:/data
    environment:
      - MONGO_INITDB_ROOT_USERNAME=emilygavin
      - MONGO_INITDB_ROOT_PASSWORD=password123
volumes:
  mydata: {}
networks:
  default:
    name: mynetwork
```

It defines a single service called "digiwallet", which is a MongoDB database instance. The service is configured to use the official "mongo" image available on Docker Hub. It exposes port 27017, which is the default port used by MongoDB, to the host machine. The environment variables "MONGO_INITDB_ROOT_USERNAME" and "MONGO_INITDB_ROOT_PASSWORD" are set to configure the root user credentials for the MongoDB instance. These credentials will be used to log in to the database and perform administrative tasks.

Figure 7.4 docker-compose file

7.e Application Properties

To connect to my database, I used the following applications properties that would initialise communication between. Here you can see my port connection, access to the database via the username and password, and the name of the collection I intend on connecting to.

```
spring.data.mongodb.host=localhost  
spring.data.mongodb.port=27017  
spring.data.mongodb.username=emilygavin  
spring.data.mongodb.password=password123  
spring.data.mongodb.database=digiwallet
```

Figure 7.5 application.properties

8. Back End Implementation

8.a Spring Boot Architecture

For my project, I used the standard Spring Boot Architecture structure.

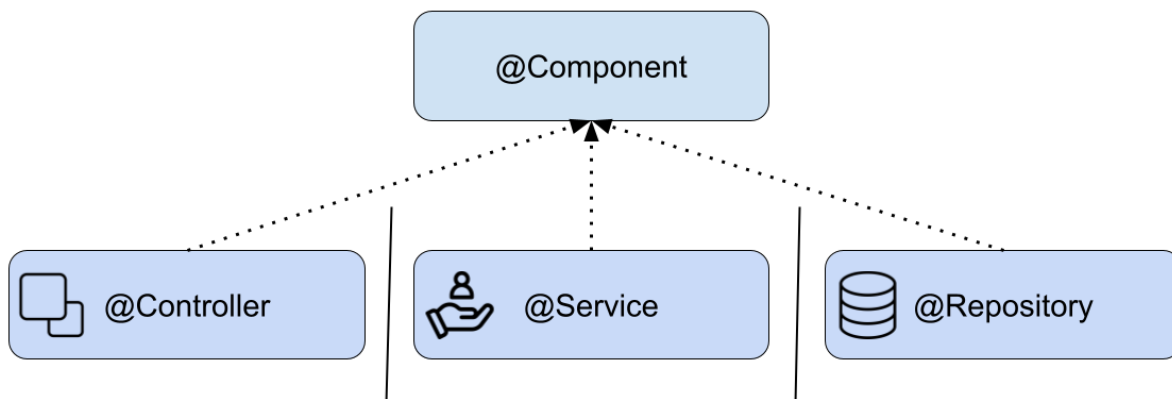


Figure 8.1 Spring Architecture Diagram

Controller layer: This is the web layer of the application. Controllers are responsible for handling incoming HTTP requests and returning appropriate HTTP responses. Controllers typically define methods that handle specific HTTP endpoints and perform the necessary logic to generate the response.

Service layer: This is the service layer of the application. Services contain the business logic of the application and are responsible for performing operations on the data. Services typically contain methods that perform specific tasks, such as adding a user, deleting a user, or retrieving a list of users from a database.

Repository layer: This layer is used to define the persistence layer of the application. Repositories are responsible for interacting with the database. Repositories typically contain methods for querying the database and returning data as Java objects. Repositories can also be used to persist data to the database using methods like `save()` or `delete()`. For my project, I extend “`MongoRepository`” which is the set of standard CRUD methods used for interacting with a MongoDB database.

8.b Controller Class

```
@RestController
@RequestMapping("api/v1/users")
```

This is a class that defines a REST API controller for managing user-related operations. It is annotated using `@Controller` and can be accessed via the general endpoint of “api/v1/users”.

Here is where I have all my end points for completing different tasks within my database. I have a mixture of CRUD methods that perform different tasks on my data, including:

fetchAllUsers(): a GET method that retrieves all the users.

fetchUser(): a GET method that retrieves a user by email and password.

registerNewUser(): a POST method that creates a new user.

AddAgeCard(), AddStudentCard(), AddDriversLicense(), AddPassportCard(): POST methods that add different types of cards to a user.

deleteUser(): a DELETE method that deletes a user by ID.

All these functions have their own unique endpoint which can be reached directly from the user to the controller layer.

At the top of my class, I created an object of the type **UserService** which I can use to communicate from my controller class through to my service layer.

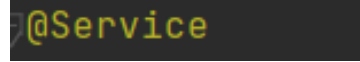
```
private UserService userService;
```

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public User registerNewUser (@RequestBody User user) throws Exception {
    return userService.addNewUser(user);
}

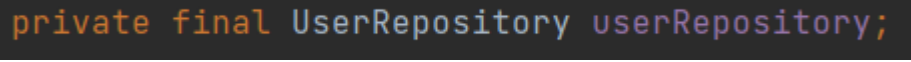
emilygavin
@PostMapping(path = "/addAgeCard/{id}")
@ResponseStatus(HttpStatus.CREATED)
public User AddAgeCard(@RequestBody AgeCard ageCard, @PathVariable String id) {
    return userService.addAgeCard(ageCard, id);
}
```

Figures 8.2 Controller Examples

8.c Service Class

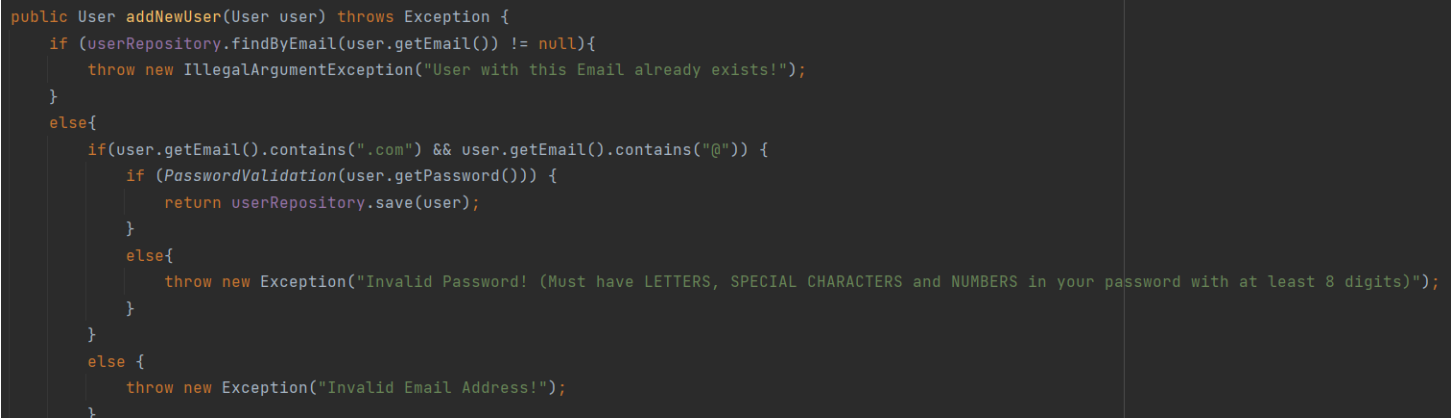


The Service class of my application is where I perform business logic on the requests between the controller layer and the repository for the DigiWallet application.



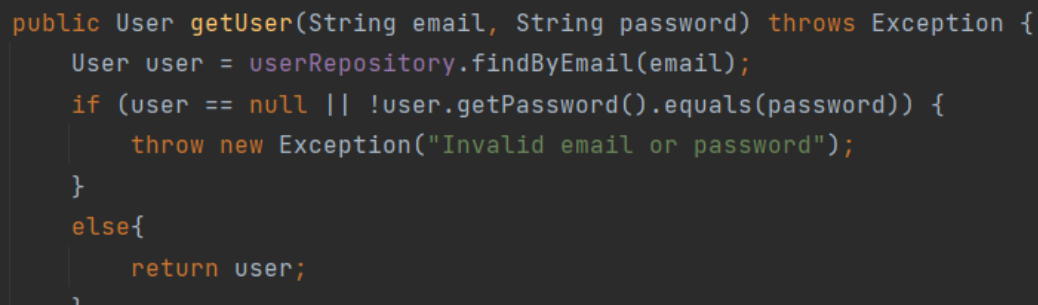
I use an object of type UserRepository to communicate from my service layer to my repository.

Examples of some business logic used within my service layer:



```
public User addNewUser(User user) throws Exception {
    if (userRepository.findByEmail(user.getEmail()) != null){
        throw new IllegalArgumentException("User with this Email already exists!");
    }
    else{
        if(user.getEmail().contains(".com") && user.getEmail().contains("@")) {
            if (PasswordValidation(user.getPassword())) {
                return userRepository.save(user);
            }
            else{
                throw new Exception("Invalid Password! (Must have LETTERS, SPECIAL CHARACTERS and NUMBERS in your password with at least 8 digits)");
            }
        }
        else {
            throw new Exception("Invalid Email Address!");
        }
    }
}
```

This code is a method called AddNewUser in a Java Spring Boot service class that adds a new user to the database. It first checks if the user's email already exists in the database, and if it does, it throws an exception with a message "User with this Email already exists!".



```
public User getUser(String email, String password) throws Exception {
    User user = userRepository.findByEmail(email);
    if (user == null || !user.getPassword().equals(password)) {
        throw new Exception("Invalid email or password");
    }
    else{
        return user;
    }
}
```

Figures 8.3 Service Examples

This code is a method called getUser in a Java Spring Boot service class that retrieves a user from the database based on the given email and password. If the retrieved user is null or the given password does not match the user's password in the database, the method throws an exception with the message "Invalid email or password". If the email and password match the user's data in the database, the method returns the user.

8.d Repository Interface

```
2 usages  👤 emilygavin  
public interface UserRepository extends MongoRepository <User, String> {  
    2 usages  👤 emilygavin  
    User findByEmail(String email);  
}
```

Figures 8.4 Repository Example

This Repository Interface called UserRepository extends the MongoRepository interface provided by Spring Data MongoDB. The MongoRepository interface provides methods for CRUD (create, read, update, delete) operations on a MongoDB database.

The UserRepository interface specifies that it will work with the User class and the ID type will be String. It also defines a method called findByEmail that takes an email address as input and returns a User object. This method is used to retrieve a user from the database based on their email address.

By extending the MongoRepository interface, the UserRepository interface inherits methods such as save, delete, findAll, etc., which can be used to perform common database operations. Since it is an interface, it needs to be implemented by a class that provides the actual implementation of these methods. Spring Data MongoDB provides this implementation at runtime.

8.e Password Encryption and Decryption

When adding a new user into my database, I ensured that methods of encryption and decryption were implemented to ensure the security of my application and its users.

Passwords are sensitive information that, if compromised, can lead to unauthorized access, identity theft, and other serious security breaches. To protect users and their data, it is essential to implement strong security measures when handling passwords.

The type of method of encryption and decryption I used for my project is the Advanced Encryption Standard (AES) symmetric-key algorithm. AES is a widely used and well-regarded encryption standard, providing a good balance of security and performance.

Encryption Key

```
encryption.key=ExampleKey123
```

I stored my encryption key into my application.properties as above to ensure further security in my encryption and decryption process.

This key is then annotated with @Bean for further use in my code.

```
@Value("${encryption.key}")
private String ENCRYPTION_KEY;

@Bean
public String encryptionKey() {
    return ENCRYPTION_KEY;
}
```

```
private final String ENCRYPTION_KEY;
```

```
public UserService(UserRepository userRepository, String encryptionKey) {
    this.userRepository = userRepository;
    this.ENCRYPTION_KEY = encryptionKey;
}
```

Encryption Method

```
public String encrypt(String password) throws Exception {
    Cipher cipher = Cipher.getInstance( transformation: "AES");
    cipher.init(Cipher.ENCRYPT_MODE, generateKey());
    byte[] encryptedBytes = cipher.doFinal(password.getBytes(StandardCharsets.UTF_8));
    return Base64.getEncoder().encodeToString(encryptedBytes);
}
```

Figures 8.1 Encryption function

This encrypt() method takes a plain-text password as input and uses the AES encryption algorithm to encrypt it. It initializes a Cipher object in the ENCRYPT_MODE and sets the secret key generated by the generateKey() method. Then, it encrypts the input password and returns the encrypted password as a Base64 encoded string.

Decryption Method

```
public String decrypt(String encryptedPassword) throws Exception {
    Cipher cipher = Cipher.getInstance( transformation: "AES");
    cipher.init(Cipher.DECRYPT_MODE, generateKey());
    byte[] encryptedBytes = Base64.getDecoder().decode(encryptedPassword);
    byte[] decryptedBytes = cipher.doFinal(encryptedBytes);
    return new String(decryptedBytes, StandardCharsets.UTF_8);
}
```

Figures 8.2 Decryption function

In contrast, this decrypt() method takes an encrypted password as input and uses the AES encryption algorithm to decrypt it. It initializes a Cipher object in the DECRYPT_MODE and sets the secret key generated by the generateKey() method. The encrypted password is first decoded using the Base64 decoder, and then it is decrypted using the Cipher object. The decrypted password is returned as a plain-text string.

9. Front End Set-up

9.a Installation

The front end of my project was completed on VSCode using Expo and React Native to create a multi-platform application for my DigiWallet project.

Expo is a free and open-source platform for building native iOS, Android, and web applications using JavaScript and React.

Before implementing my front-end code, I had to ensure I had the expo library installed on my machine and from then I created a new expo project. This was made possible using the following commands:

```
npm install -g expo-cli
```

```
expo init DigiWallet
```

Ngrok is a service that allows you to expose a web server running on your local machine to the internet. I used Ngrok so that I could get backend localhost endpoints and tunnel them to make it possible for my database information to be available for view to my front-end application. This would not be possible without either using a tunnel or using a cloud database system.

The easiest way to achieve Ngrok connection is via a script. Here's an example of the script:

```
ngrok (Ctrl+C to quit)
Announcing ngrok-rs: The ngrok agent as a Rust crate: https://ngrok.com/rust

Session Status      online
Account             Emily Gavin (Plan: Free)
Update              update available (version 3.2.2, Ctrl-U to update)
Version             3.1.1
Region              Europe (eu)
Latency              39ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://c1b9-2001-bb6-464-c300-acc9-328b-d56d-a182.ngrok-free.app -> http://localhost:8080

Connections
  ttl    opn    rt1    rt5    p50    p90
   0      0     0.00   0.00   0.00   0.00
```

Figures 9.1 Expo and Ngrok

Here you can see that I forward my localhost address <http://localhost:8080> (which is MongoDB's default localhost address) to a new Ngrok hosted address. To then access my endpoints from my backend application from my front-end, I can simply use the Ngrok forwarding address, and this will then access my database within my computer.

10. Front End Implementation

10.a Native Stack Navigator

The main navigation pages for my project are done using the React Navigation library. The code sets up a stack-based navigation system, defining different screens in the app and how they are connected.

Inside the `<Stack.Navigator>` component, multiple `<Stack.Screen>` components are defined. Each of these components represents a screen within the application, which include Home, Log in, Sign up, and Welcome screens along with much more. The name prop assigns a unique identifier to each screen, while the component prop specifies the React component that should be displayed when that screen is active.

Each Screen component has an options prop, which is used to customize the appearance and behavior of the screen, such as whether the header should be shown or hidden, and the styling of the header.

Each screen is associated with a specific React component that will be rendered when the user navigates to that screen.

```
const Stack = createNativeStackNavigator()

export default App = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          options={{headerShown: false}}
          name="Home"
          component={HomeScreen}
        />
        <Stack.Screen
          options={{headerShown: false,
            headerStyle: {
              backgroundColor: '#2c365a',
            },
          }}
          name="Log In"
          component={LogInScreen}
        />
        <Stack.Screen
          options={{headerShown: false,
            headerStyle: {
              backgroundColor: '#2c365a',
            },
          }}
          name="Sign Up"
          component={SignUpScreen}
        />
        <Stack.Screen
          options={{headerShown: true}}
          name="Welcome Screen"
          component={WelcomeScreen}
        />
        <Stack.Screen
          name="Fetch"
          component={FetchScreen}
        />
      </Stack.Navigator>
    </NavigationContainer>
  )
}
```

Figures 10.1 Native Stack Navigator

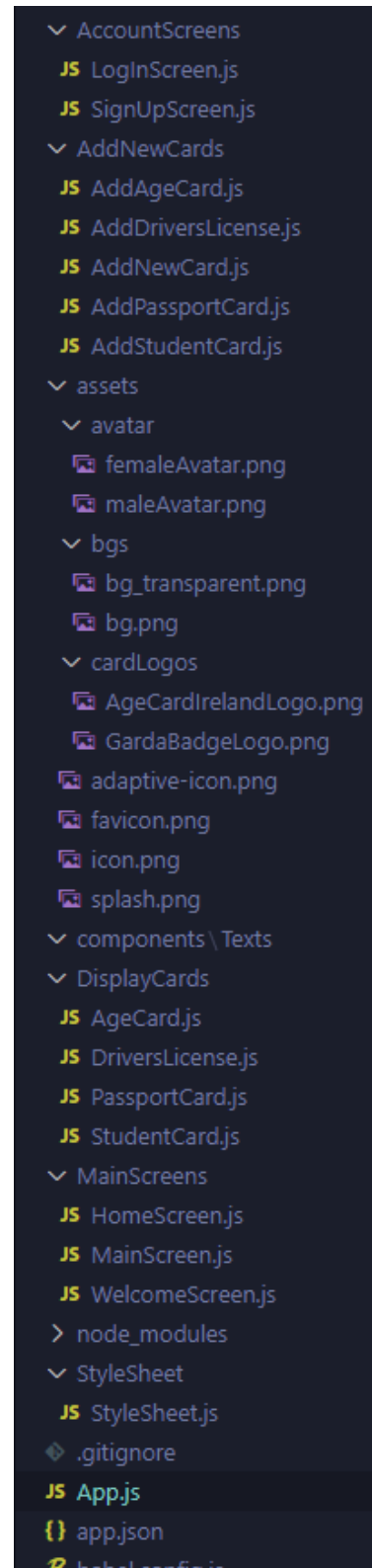
10.b Folder Organization

Within my project there are many different screens, so project organization was very important to ensure clean, and easy to manage code.

Folders in my front-end application are based on their functionality or role within the application. Consistent naming for files and folders makes it easier to identify the purpose of each component or module.

Some of these folder names include AccountScreens, AddNewCards, DisplayCards, MainScreens and StyleSheet.

I also file my photos away into their separate folders underneath the assets folder. This ensures all my files are kept exactly where I know to find them.



Figures 10.2 Folder Organization

10.c User Screens

Log In Screen

When the user initially opens the application, they are prompted to either Log in or Sign up. A login or sign-up screen is an essential part of many applications, providing various benefits for both users and developers.

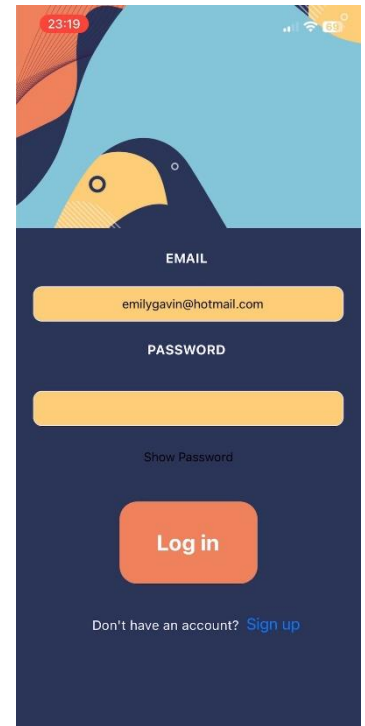
When logging in, the user hits the following endpoint:

```
const callAPI = async () => {
  try {
    const res = await fetch(URL + `/api/v1/users/login?` + new URLSearchParams({
      email: email,
      password: password
    })),
  },
}
```

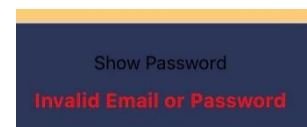
This endpoint will hit my backend which will ensure that the users exists by email, and then will check to see if the password entered matches the email that has been found. If not the user will receive a haptic feedback buzz along with the following error message:

```
} else {
  setErrorMessage('Invalid Email or Password');
  Vibration.vibrate(500); // Add haptic feedback
}
```

The reason I use the phrase “Invalid Email or Password” is because providing more specific error messages like "Invalid email" or "Incorrect password" can give an attacker information that they can use to guess the correct login credentials. By keeping the error message generic, it makes it harder for an attacker to determine which part of the login credentials is incorrect.



Figures 10.3 Log in Screen



Sign Up Screen

For the Sign-up page in my application, more information is needed to create a new account. This is because when a new user is added to the database, there is more information needed to fulfil the necessary User body within my backend endpoint.

When a new user enters all their details, they are then redirected to take an ID picture.

This ID picture appears on all the users ID cards to identify the user when the ID cards are being used.

The image is stored within the database under the variable “uri” and when the user successfully has filled out all the necessary values, the new user is the posted to the userbase with a empty card main screen for the user to add to.

Further Security

There are several ways in what I could further implement to make this sign-up process more secure. This includes:

- Verify the email address of new users by sending them a unique link or code to confirm their email. This helps to prevent fake or disposable email addresses from being used.
- Have users verified using a passport photo to ensure that the user matches the information they provided when signing up.
- The use of secure connections. Requiring users to sign up over a secure connection (HTTPS) helps to ensure that their information is protected against eavesdropping and

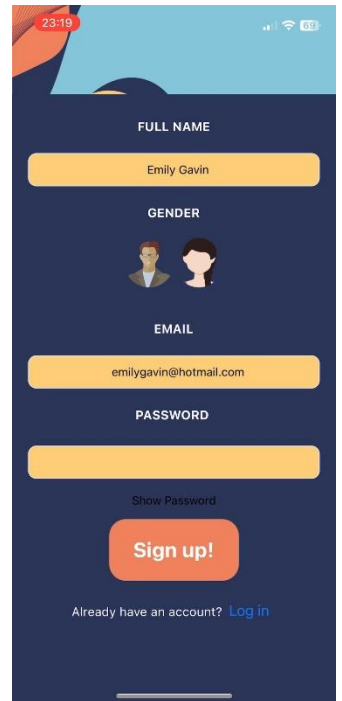


Figure 10.4 Sign Up Screen

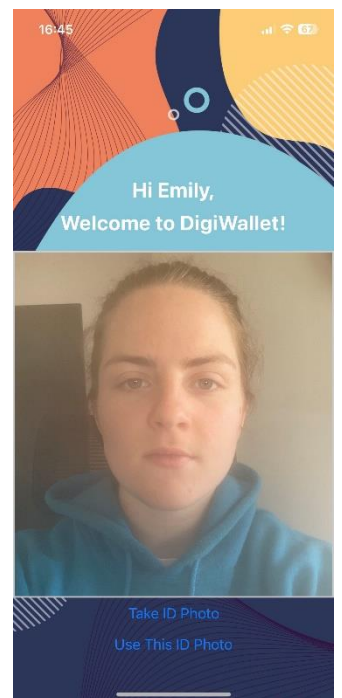


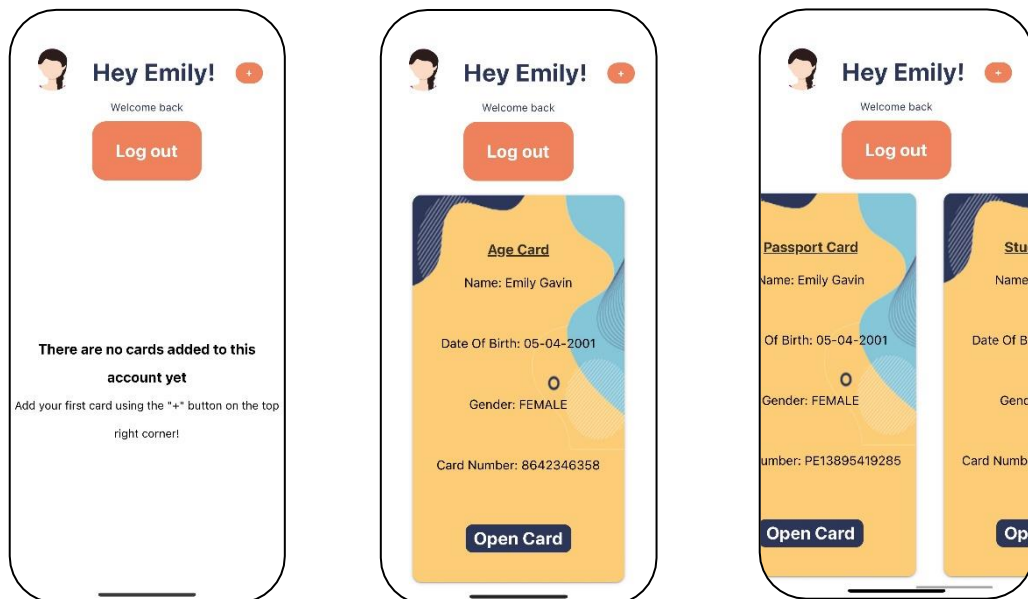
Figure 10.5 Welcome Screen

10.d Main Screen

After the user has either logged in or sign-up, they will be introduced to the DigiWallet Main screen. Here the user can browse the cards on their account, add new cards, or click into a card to view their full card information. I have added many checks on this screen so that the user has a seamless and aesthetically pleasing UI experience when working with the application. Some examples of this on my main screen include:



A small graphic with the users chosen gender will appear along side the users first name.



Cards will only appear when they have been added to an account, so if you have only added 2 cards, only 2 cards will appear. This is made possible by the following code:

```
const renderItem = ({ item }) => {
  if (item.typeOfCard) { // Add a co
    return (
```

```
} else {
  return null; //
}
```

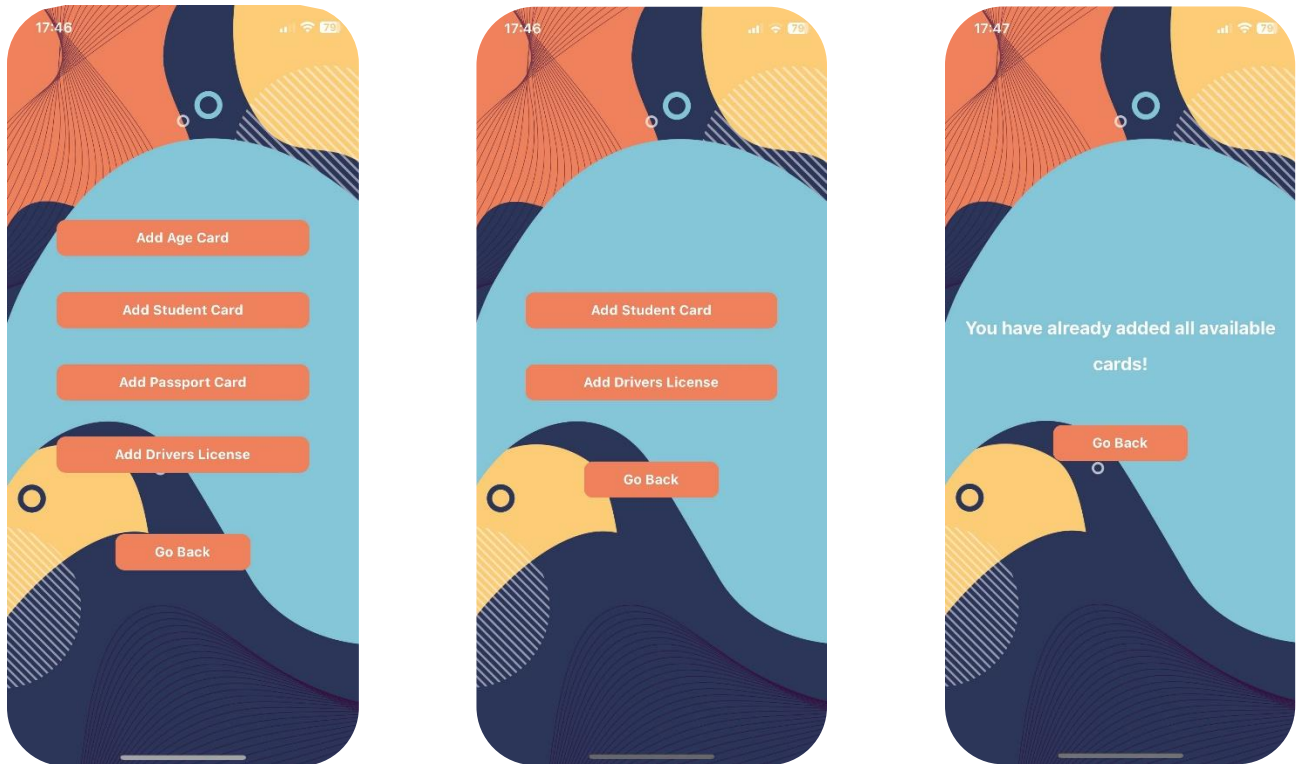


Figure 10.6 Add New Card Screen

The adding of a card to a user's account can only be done if the card has not already been added to the database. When the user has added all cards that are available on my application, they will be prompted with text saying, "You have already added all available cards!". This is made possible by the following code:

```
const isAllCardsFilled = cardData.ageCard !== null && cardData.studentCard !== null && cardData.passportCard !== null && cardData.driversLicense !== null;
```

I created a const called isAllCardsFilled which checks to see if all cards are filled.

```
{!isAllCardsFilled && cardData.ageCard === null && (
```

Then for the addition of each card, I check my isAllCardsFilled field, and check if the card is null.

```
{isAllCardsFilled && (
  <Text style={[styles.smallTextBold, { color: "white"}]>
    You have already added all available cards!
  </Text>
```

If the isAllCardsFilled field is then full, it will print the error message to the user.

10.e Card Screen

The title of the card is displayed, followed by the users ID photo.

The user's variables are displayed, each card has their own unique set of variables.

An accelerometer is implemented which changes colour depending on the phones x, y, z values. This proves that the card has not been screenshotted, and that the user's card is legit upon viewing.

A Go Back button to return the user to the main DigiWallet screen.

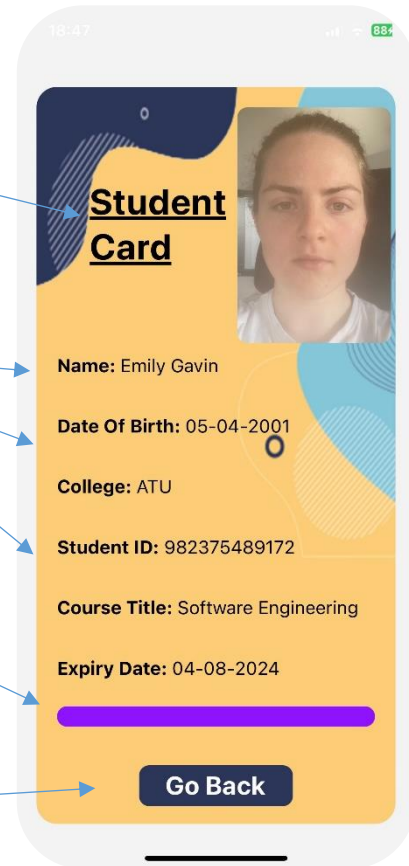
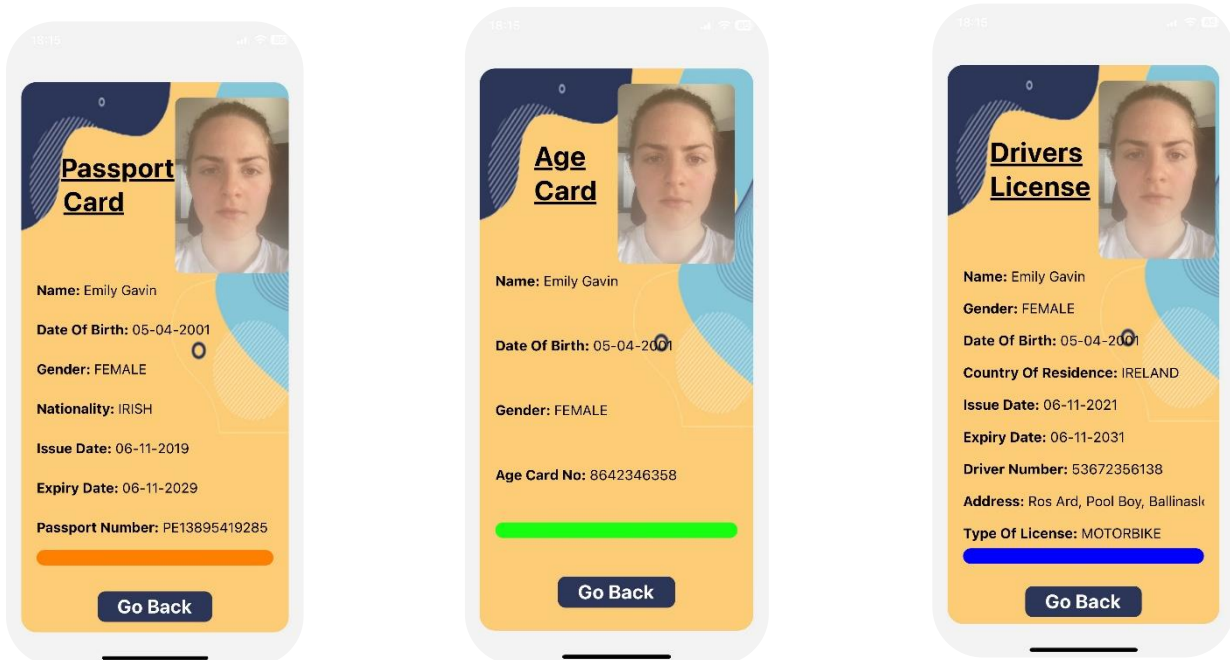


Figure 10.7 Card Screen



10.f Main Screen Progression Example



Figures 10.8 Main Screen Cycle

11. Cloud Connection

11.a Mongo Atlas

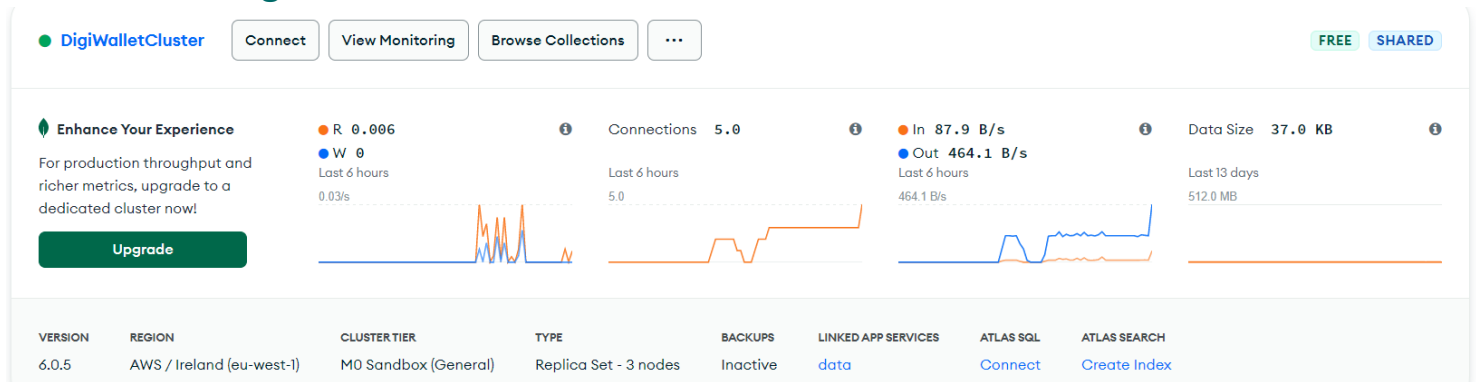


Figure 11.1 Mongo Atlas Cluster

My DigiWallet database is connected to Mongo Atlas. I did this by creating a cluster on Mongo Atlas and connecting my backend via a connection string as per the following:

```
spring.data.mongodb.uri=mongodb+srv://emilygavin:Password123@digiwalletcluster.ehafqae.mongodb.net/digiwallet
```

This code was added to my application.properties in my back-end spring boot project and by entering my MongoDB username, password, cluster name, and database name, I could connect to my existing cluster. By connecting my database to the Mongo Atlas cluster, my data is now available from a cloud, meaning that data could be received and sent to the cluster without the need for a back-end application to always be running. It also ensures a much safer transfer of data to and from the database as there are multiple security checks that are needed before a HTTP request could be made to the database. Here is an example of postman connecting to the cloud database:

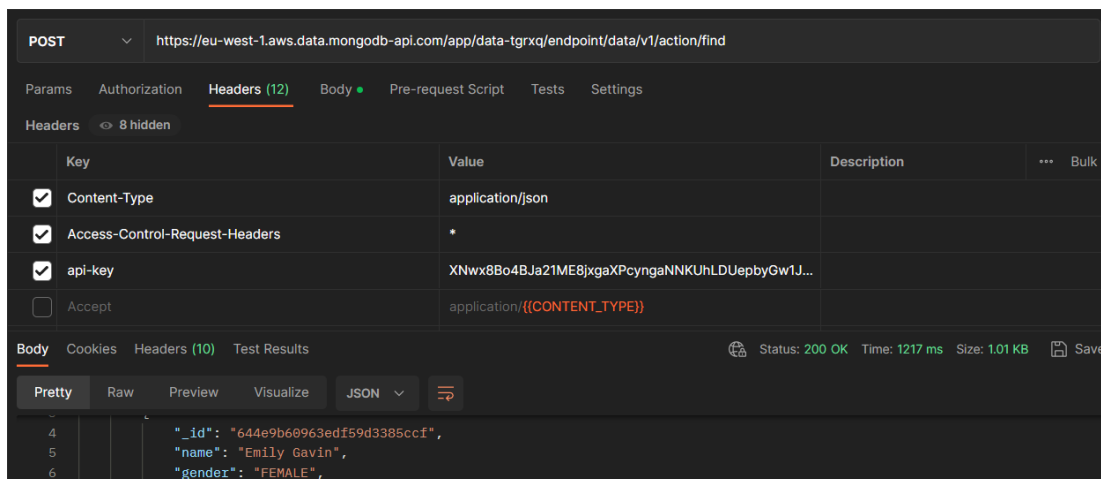
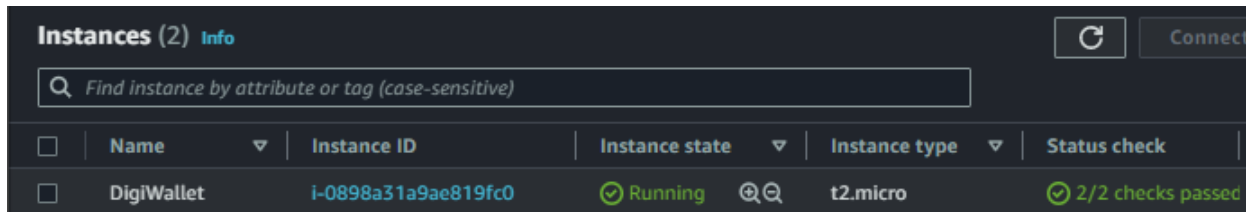


Figure 11.2 Postman Request

11.b AWS EC2 Instance

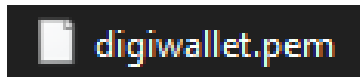
I have my backend application connected to AWS to host my endpoints on a EC2 instance. This is important to have running so that my project can be available on a cloud with the need for local connection. This was completed with the following steps:

1. Set up a new instance and configure as necessary



Instances (2) Info						
Find instance by attribute or tag (case-sensitive)						
	Name	Instance ID	Instance state	Instance type	Status check	
<input type="checkbox"/>	DigiWallet	i-0898a31a9ae819fc0	Running	t2.micro	2/2 checks passed	

2. Create a .pem file



I had to store my .pem in a secure location on my computer and this was an import security step for connecting my backend project to my instance

3. Connect to my Instance on EC2 Instance Connect

This opens a terminal page on my Ubuntu Instance where I can connect to my files on my location computer and run a file on the instance.

4. Create a jar file of my backend application and move to the instance so I can run my application on EC2 remotely

To create my .jar file of my project, I used the command: `mvn clean package`.

5. Connect my .jar file to my .pem and ec2 public IP.

To do this, I used to the following command:

```
scp -i /path/to/your-private-key.pem your-app-name.jar ec2-user@your-ec2-public-ip:/home/ec2-user/
```

6. Finally, run my application through my Instance for a fully functioning cloud backend application.

Here is an example of my project running on my instance:

```
ubuntu@ip-172-31-21-246:~$ sudo java -Dserver.port=80 -jar DigiWallet-0.0.1-SNAPSHOT.jar

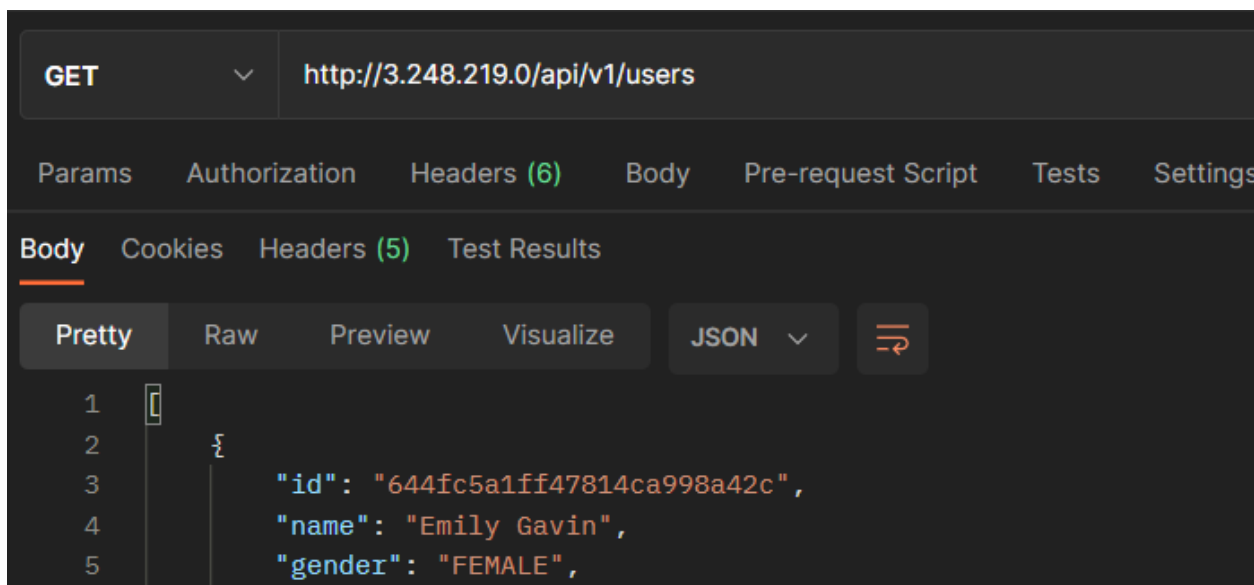
:: Spring Boot ::
(v2.4.5)

2023-05-03 12:31:16.476 INFO 6777 --- [main] c.e.DigiWallet.DigiWalletApplication : Starting DigiWalletApplication v0.0.1-SNAPSHOT using
2023-05-03 12:31:16.490 INFO 6777 --- [main] c.e.DigiWallet.DigiWalletApplication : No active profile set, falling back to default profil
2023-05-03 12:31:18.090 INFO 6777 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data MongoDB repositories in DEF
2023-05-03 12:31:18.202 INFO 6777 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 98 ms. Fo
2023-05-03 12:31:19.284 INFO 6777 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 80 (http)
2023-05-03 12:31:19.313 INFO 6777 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-05-03 12:31:19.314 INFO 6777 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.45]
2023-05-03 12:31:19.443 INFO 6777 --- [main] o.a.c.c.C.[Tomcat].[/] : Initializing Spring embedded WebApplicationContext
2023-05-03 12:31:19.448 INFO 6777 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
2023-05-03 12:31:19.914 WARN 6777 --- [main] o.s.h.c.j.Jackson2ObjectMapperBuilder : For Jackson Kotlin classes support please add "com.f
2023-05-03 12:31:20.347 INFO 6777 --- [main] org.mongodb.driver.cluster : Cluster created with settings (hosts=[127.0.0.1:27017
t='30000 ms', requiredReplicaSetName='atlas-ylauxu-shard-0')
2023-05-03 12:31:20.381 INFO 6777 --- [gae.mongodb.net] org.mongodb.driver.cluster : Adding discovered server ac-0clvotf-shard-00-02.ehafa
2023-05-03 12:31:20.488 INFO 6777 --- [gae.mongodb.net] org.mongodb.driver.cluster : Adding discovered server ac-0clvotf-shard-00-01.ehafa
2023-05-03 12:31:20.492 INFO 6777 --- [gae.mongodb.net] org.mongodb.driver.cluster : Adding discovered server ac-0clvotf-shard-00-00.ehafa
2023-05-03 12:31:21.625 INFO 6777 --- [ngodb.net:27017] org.mongodb.driver.connection : Opened connection [connectionId{localValue:3, serverV
2023-05-03 12:31:21.626 INFO 6777 --- [ngodb.net:27017] org.mongodb.driver.connection : Opened connection [connectionId{localValue:2, serverV
2023-05-03 12:31:21.628 INFO 6777 --- [ngodb.net:27017] org.mongodb.driver.cluster : Monitor thread successfully connected to server with
```

Figure 11.3 EC2 Instance Connect

These endpoints can then be successfully accessed through the public IP address of my EC2 instance.

Here is an example of my endpoints being hit through postman:



12. Testing

12.a Unit Testing

In the development of the DigiWallet application, ensuring the reliability and stability of the software is crucial. One of the best practices to achieve this is by using unit testing. Unit tests are designed to test individual units or functions of an application, making it easier to find and fix errors/bugs as they arise during development. The following report provides an overview of the unit tests implemented in the DigiWallet application and the importance of unit testing in the software development lifecycle.

In the DigiWallet application, I wrote a series of tests for the UserService class, which interacts with the UserRepository to manage user data. The tests cover various scenarios such as getting all users, handling invalid email addresses, adding new users with different constraints, and verifying password constraints. These tests ensure that the UserService class behaves as expected, and any changes to the code do not introduce unintended effects.

I tested my backend development using Junit and used Mockito to mock my data within a fake database. Here is an example of some of my Unit Tests for my UserService class:

```
@Test
public void testAddNewUserInvalidEmail() {
    User user = new User( name: "John", gender: "Male", email: "invalid_email", password: "Password!1
    when(userRepository.findByEmail("invalid_email")).thenReturn( t: null);

    assertThrows(Exception.class, () -> userService.addNewUser(user));
}

emilygavin
@Test
public void testAddNewUserInvalidPassword() {
    User user = new User( name: "John", gender: "Male", email: "john@gmail.com", password: "password"
    when(userRepository.findByEmail("john@gmail.com")).thenReturn( t: null);

    assertThrows(Exception.class, () -> userService.addNewUser(user));
}
```

Figure 12.1 Unit Testing Examples

13. Future Development

During the course of completing this project, I came up with a few ideas that I feel could greatly bring on the DigiWallet Application. While I am quite pleased with the progress made within the given timeline, I'd like to share some additional concepts that could be considered for future development:

- **PPS Confirmation:** The connection of a user's PPS that could check what cards may be available for them to add to their DigiWallet application. i.e. If a user has not yet turned 18, an age card would not be available for the user to add to their account. This would need some database connection with government information.
- **Automatic Camera Input.** It would have been interesting to add some functionality that the user could take a photo of their desired ID card and that the information would automatically add to their account. This would use camera edge detection technology to pick up card information.
- **Flippable cards.** When designing the cards, I realized that some cards need two sides of information available, i.e. Driver's License. It would have been a cool addition to the viewing of the cards but unfortunately, I never got the time to implement this feature.
- **ID Photo Verification:** As of now in my project, there is no photo verification that checks to see if the photo the user took is an accurate and clear photo of the user. It would be important that this photo is to be checked by a system or human to ensure that the photo is: not blurry, the users eyes are open, the user is not smiling etc.

14. Conclusion

In conclusion, I am highly satisfied with how my project turned out. It struck a good balance between knowledge I had previously gained through my internship and college experiences, and integrating new material as I progressed through the project. This learning process has been both challenging and rewarding.

The back-end development of the project, using Java and Spring, proved to be an area that I really enjoyed the completion of. It has solidified my interest in this type of Software Engineering, and I think that I will be trying to get into this area of expertise after college.

The front-end development proved to be a significant learning curve, which initially posed some challenges. However, as the project progressed, I gained confidence in my front-end skills and expanded my understanding of creating seamless user experiences. I am sure this knowledge will serve me well in future endeavours.

One of the primary aims of my project was to demonstrate the potential for ID cards to become more sustainable through the use of digital technology. The successful completion of this application illustrates that such a transition not only very possible but could also make a huge impact on our environment. By implementing small yet meaningful changes like this, we can make a significant positive impact of our footprint on the plant.

As I come to the completion of this project, I feel a sense of accomplishment in how far I have come since September. I have developed a functioning full-stack application with all the functionality I planned to implement. I have grown as a developer and have broadened my expertise in Software Engineering. I feel more prepared and confident in my ability to transfer my software skills to the working world and to future projects I encounter.

15. References

[1] <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring->

boot/#:~:text=Java%20Spring%20Boot%20is%20an,computing%20platforms%20for%20app%20development.

[2] [https://earthly.dev/blog/mongodb-](https://earthly.dev/blog/mongodb-docker/#:~:text=If%20you%20want%20to%20use,than%20manually%20configure%20a%20server.)

docker/#:~:text=If%20you%20want%20to%20use,than%20manually%20configure%20a%20server.

[3] <https://www.youtube.com/watch?v=ayTPvC7iikh>

[4] <https://amigoscode.com/p/full-stack-spring-boot-react>

[5] <https://www.youtube.com/watch?v=7-XgjelXvQ>

[6] ChatGPT was used at various points of this report to enhance the clarity and overall quality of the writing. ChatGPT was used to refine some paragraphs, making it more clear and engaging for the readers. This allowed the report to effectively communicate the purpose, design, and implementation of the DigiWallet project, ensuring that readers could easily understand the project's goals, technologies used, and the challenges faced during its development.