S.H.A.R.E
Final Report

1. Introduction

1.1 Purpose of the System

The system is a borrowing and lending platform for users to borrow and lend items within the same vicinity or community, upon agreement. The system aims to provide and serve as a model solution to the real-life problem pertaining to students in student dormitories having limited access to appliances and utilities due to limited resources: spatial capacity, funds, or time.

1.2 Definitions and Acronyms

S.H.A.R.E. = Student Housing Appliance Rental Exchange
UML = Unified Modeling Language

2 Software Requirements Specifications

2.1 Functional Requirement

The functional requirements have been listed in Table 2-1.

Process Oriented	2.1.1 The system shall allow users to sign up for an account.
	2.1.2 The system shall ask users to authenticate themselves.
	2.1.3 The system shall ask users to specify their location when signing up.
	2.1.4 The system shall allow users to sign into their account.
	2.1.5 The system shall allow users to change their password if it was

	forgotten.
	2.1.6 The system shall allow users to list an item for lending.
	2.1.7 The system shall require the lender to upload a photo of the item to be lent
	2.1.8 The system shall allow users to specify the duration an item is available.
	2.1.9 The system shall allow users to search an item.
	2.1.10 The system shall allow users to make a borrowing request for an item.
	2.1.11 The system shall allow users to specify the duration an item is needed.
	2.1.12 The system shall require the users to provide a payment method.
	2.1.13 The system shall allow users to accept/reject a borrowing request
	2.1.14 The system should notify the user before the borrowing duration is over.
	2.1.15 The system shall ask the users to both confirm when an item has been returned.
	2.1.16 The system shall ask the user if the item was returned damaged or in good condition.
	2.1.17 The system shall hold the price of the item throughout the borrow lend process.
	2.1.18 The system shall return the amount to the borrower if the item is not damaged.
	2.1.19 The system shall send the money to the user if the item is

	damaged.
	2.1.20 The system shall allow the user to request to borrow an item of their specification
Information Oriented	2.1.21 The system shall store the users' data.
	2.1.22 The system shall store all currently listed items.
	2.1.23 The system shall store the list of items lent and borrowed by the user.
	2.1.24 The system should record the percentage of items damaged by each user.
	2.1.25 The system shall record the rating of the user.
	2.1.26 The system shall store the users payment information.

Table 2-1 Functional Requirements

2.2 Non-functional Requirements

The Non-functional requirements have been listed in Table 2-2.

Performance	2.2.1 The system shall operate 23 hours a day.
	2.2.2 The system shall support 1000 users at any given time.
	2.2.3 The system shall respond within 5 seconds for authentication, borrowing and payment processing
	2.2.4 The system shall function on real time processing.
	2.2.5 The system shall be equipped to

	handle an increasing number of users and listings without having any significant impact on performance.
Operational	2.2.6 The system shall work on IOS.
	2.2.7 The system shall work on android.
Security and safety	2.2.8 The system shall store sensitive information in a secure way.
	2.2.9The system shall encrypt user's sensitive information so that it is not available for anyone, including administrators.
	2.2.10 The system shall follow the GDPR regulations.
Usability	2.2.11 The system shall have a consistent design to ensure that the actions performed by a user are repeatable and predictable.

Table 2-2 Non-Functional Requirements

2.3 Prioritised Functional and Non-functional Requirements

Functional Requirements (High Priority):

The system shall allow users to list an item for lending.
The system shall allow users to make a borrowing request for an item.
The system shall hold the price of the item throughout the borrow lend process.
The system shall allow users to accept/reject a borrowing request

Non-Functional Requirements (High Priority):

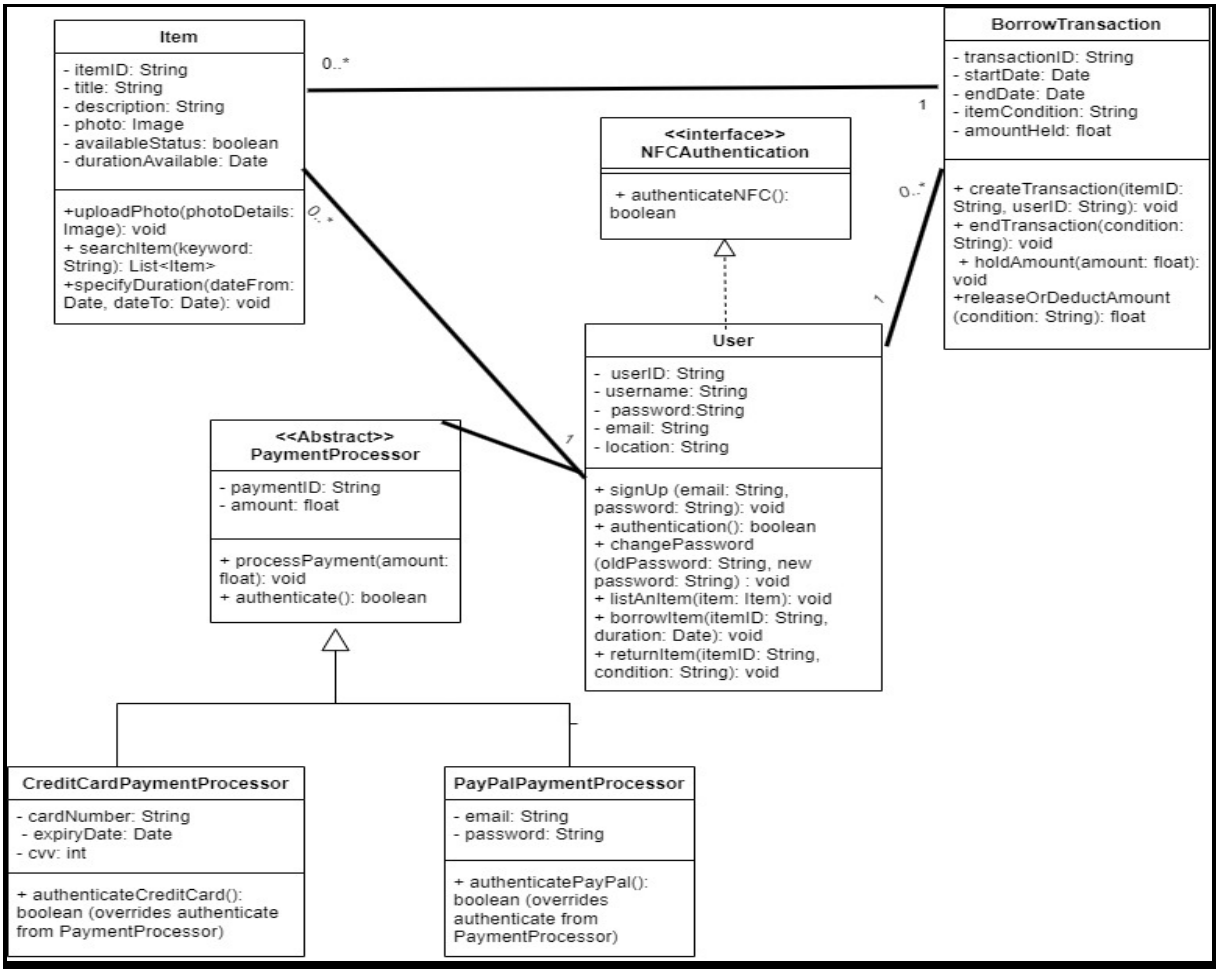
The system shall respond within 5 seconds for authentication, borrowing and payment processing
The system shall encrypt user's sensitive information so that it is not available for anyone, including administrators.
The system shall have a consistent design to ensure that the actions performed by a user are repeatable and predictable.

3. Software Design Specifications

3.1 UML Use Case Diagram

The use case diagram above displays the main functionality and actions that the relevant actors, in this case, the users and the system administrator can complete using the S.H.A.R.E. application. The users should be able to register for an account and login to this account. They should also be able to authenticate using NFC, and add a payment method before borrowing or lending a listed item. In addition, users should be able to search, borrow, list, and borrow items using the application, as well as creating or starting a transaction and ending a transaction, upon borrowing or lending an item. Consequently, the system administrator should be able to view all the transactions being made using S.H.A.R.E., manage the items being borrowed and lent, as well as set the system parameters and be able to maintain the application.

3.2 UML Class Diagram



The class diagram for S.H.A.R.E contains:

1. Five classes: User, Item, BorrowTransaction, CreditCardPaymentProcesser,and PayPalPaymentProcesser.
2. One abstract class: PaymentProcesser.
3. One interface: NFCAuthentication.

The User class contains the user´s ID, name, password, email and location. It has methods for signing up, authentication, changing the password, listing or adding an item, borrowing an item, and returning an item.

The Item class contains the item´s ID, title, description, photo, availableStatus, and durationAvailable.It has methods for uploading a photo, searching an item, and specifying duration.

The BorrowTransaction class contains the transaction´s ID, startData, endDate, itemCondition, and amountHeld. It has methods for creating a transaction, ending a transaction, holding an amount of money, and releasing or deducting that amount.

The PaymentProcessor abstract class contains the payments ID and the amount. It processes the payment and authenticates the payment method. This will be implemented as an Abstract class to allow flexibility, so if more payment methods are added to the application in the future they can be easily implemented as sub clases.

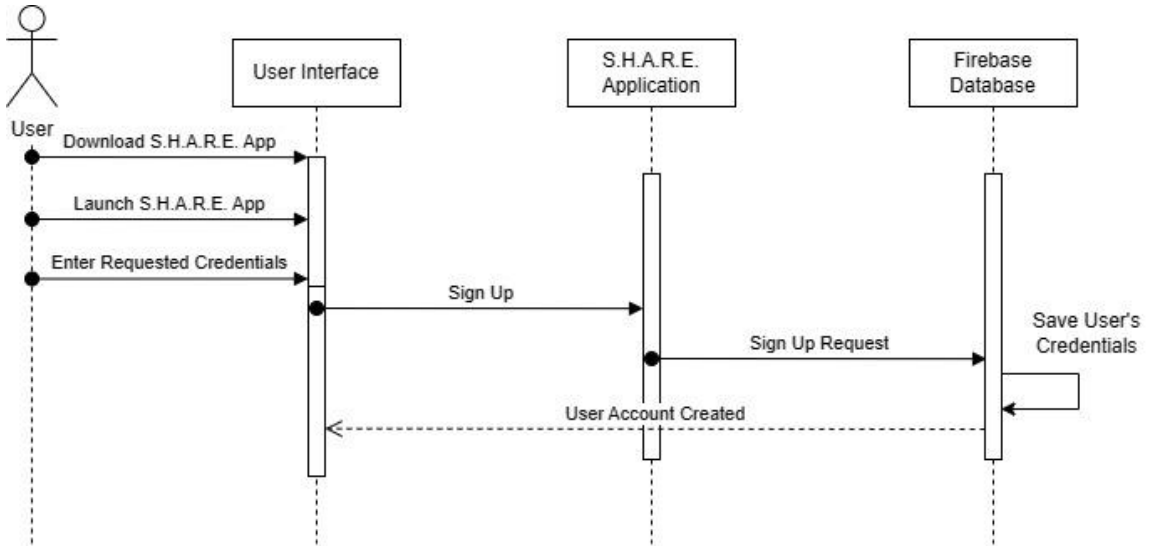
The CreditCardPaymentProcesser is a subclass of the PaymentProcessorand it contains the card number, expiry date, and ccv. It authenticates the credit card.

The PayPalPaymentProcesser is a subclass of the PaymentProcessor and it contains the email and password. It authenticates the paypal account.

The NFCAuthentication interface is responsible for user authentication. The decision to make it an interface rather than an abstract class was made so if other authentication methods are added in the future the User class would be able to inherit from all of them rather than inheriting from one abstract class.

3.3 UML Sequence Diagram

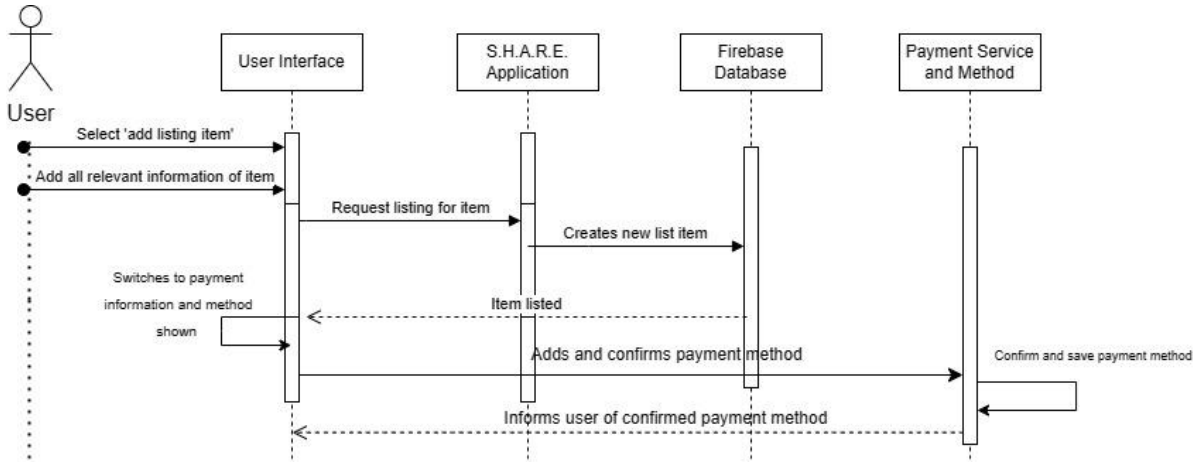
3.3.1. Sequence Diagram 1: Signup Process



The sequence diagram above illustrates the process of signing up for an account with S.H.A.R.E. First, because the user needs to download the S.H.A.R.E. application on their designated device, and then launch the application. Upon launching the app, the user will open the app to a sign up/login page asking the user

to sign up, or login if they already have an account. The user will then enter their preferred email and password. Once the user enters this information, the user interface will send a signup message to the application, indicating an account is trying to be made. The application will then send a signup request to the database for the database to save and create the user's account and store it. The database will then return a message or some form of indication to the user interface notifying them the account has been created.

3.3.2. Sequence Diagram 2: Listing An Item Process

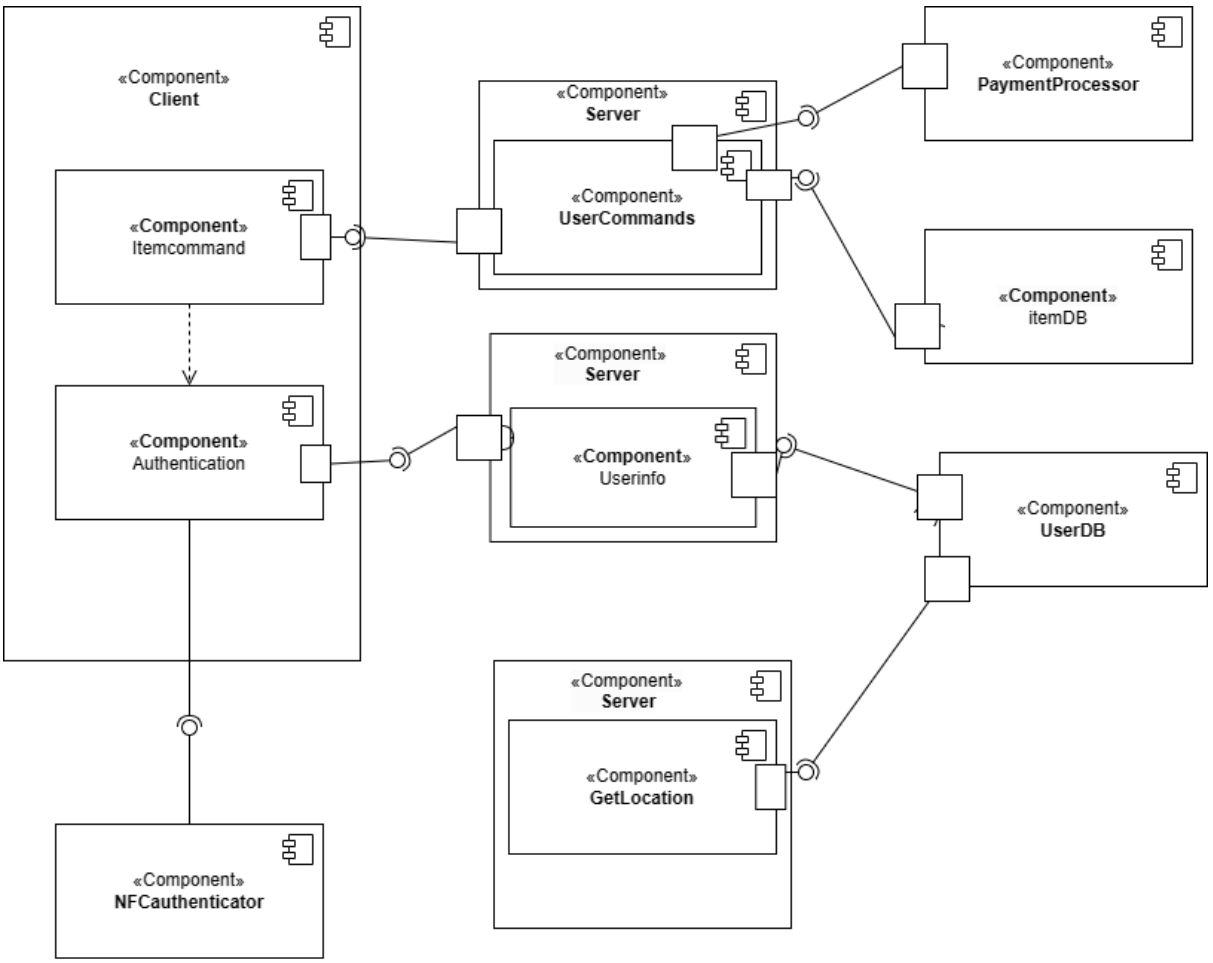


The sequence diagram above illustrates the process of a user listing an item on S.H.A.R.E. for people to borrow/request to borrow. First, the assumptions made are that the user has downloaded the app, the user has opened the app, and the user has logged into their account. At this point, the user will select the option "add listing item". Upon selecting this, the user will then add all the relevant information and description of the item being listed. Once the user enters this information, the user interface will send a request message to the application, indicating an item is trying to be added to be listed, along with its corresponding information. The application will then send a further request to the database for the database to add and save the item. The database will then return a message or some form of indication to the user interface notifying them the item has been listed. After the item is listed, the user interface will change to a payment method message asking the user to put a payment method in case the item is destroyed or damaged, so the user can collect the agreed or listed amount in case of damage. Once the payment method is added, then it is connected to the payment service to confirm and save the payment method. Once this happens, then a message to the user displayed by the user interface will indicate the confirmation of the payment method.

3.4 UML Component Diagram

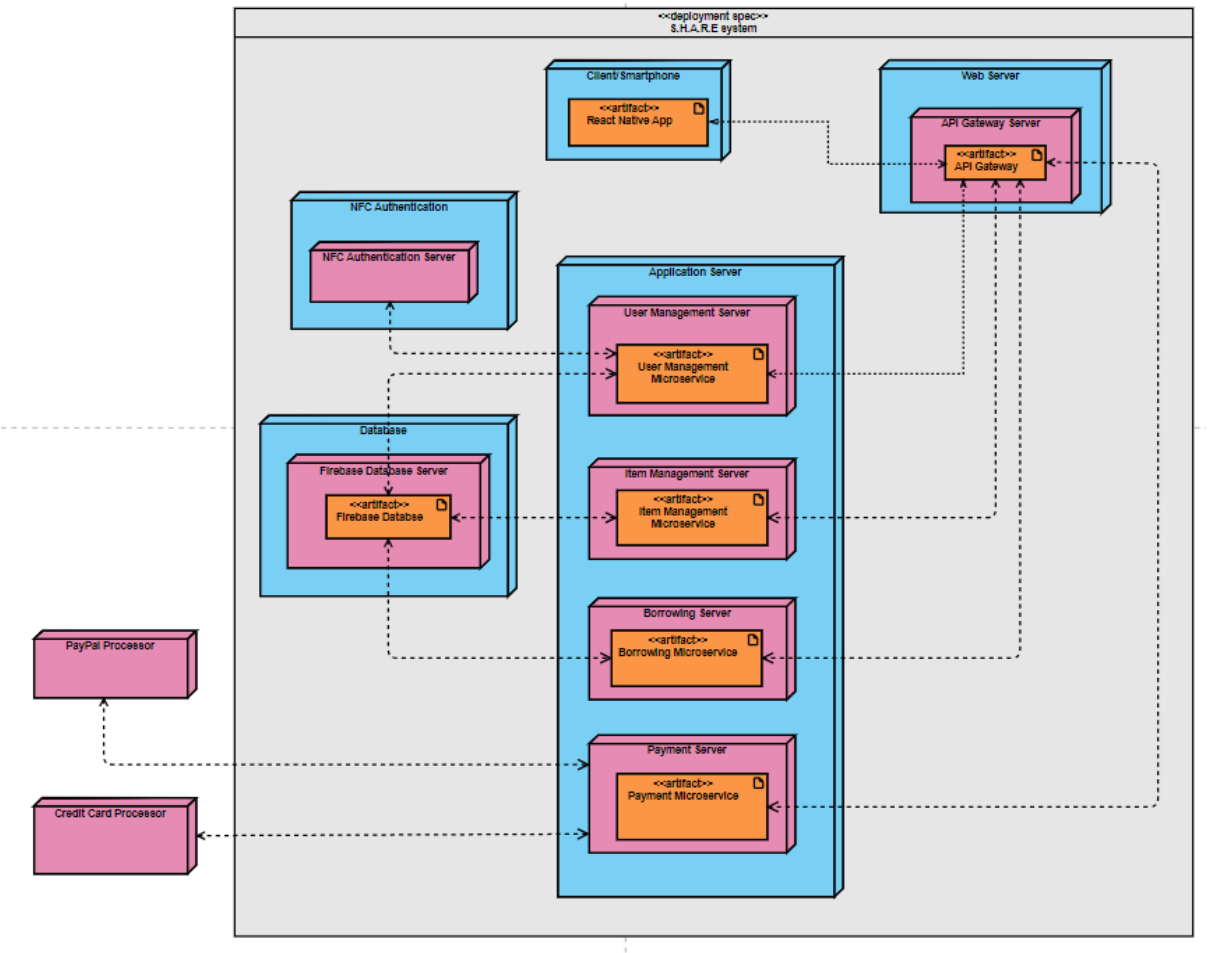
A component diagram describes the organization and wiring of a system's main parts.

This component diagram is designed to support abstraction and separation of entities. The presence of multiple separate components in the program ensures abstraction and flexibility. For instance, the separation of the item database, the user database, and the PaymentProcessor ensures code abstraction and flexibility.This makes them independent from each other, meaning that they can be edited/ replaced without affecting each other's functionality. One more positive of this architecture is that it ensures scalability, with the increase of clients on the application, multiple servers can be used and almost no change will be required in the system. Moving on to the diagram itself, it is split into six main components. Those components are Client,Server, Item DB, User DB, NFC authenticator and payment processor. The NFC authenticator communicated directly with the client's subcomponent authenticator in order to authenticate the user. This authenticator then creates a port that includes all user information then forwards it to the Userinfo Subcomponent in the server. Userinfo then takes the port with those commands and creates a new port that includes the request to be done in the UserDB. If the DB is successful with the command it does not reply. If it is not successful it replies with an error message. The database does not reply when it is successful for performance purposes. Once this process is done, the client starts a client session. In the client session, one of many item commands can be done. Those item commands are taken, put into a port and get sent to the performcommand subcomponent in the Server. The performcommand then takes it, creates two new ports and sends the first one as a request to the ItemDB component and the paymentprocessor component. To send the request to the DB, it creates a port, and sends it to the itemDB the command it wants to do. If the command is successful, no reply is given. If it fails, a reply with an error message is given. It sends the second port to the payment processor so it can process payment. If the payment is successful, then the payment processor won't reply. If not, a reply will be sent to the client stating the problem. One more action that the performcommand does is that it makes a request to GetLoction sucomponent in the Serve so it can check if the user is eligible to use the app from the requested location. The GetLocation sends to the database and checks the location that the user should use the application from. It then starts checking the user's location, if the user is in the correct place, a request can be made. If not, the user will not be able to create the request unless they are in the desired location.



3.5 UML Deployment Diagram

.



The deployment diagram for S.H.A.R.E illustrates the software and hardware components and how they are distributed across various nodes.

A web server is responsible for handling HTTP requests and responses. It primarily deals with serving static content like HTML, CSS, JavaScript, and media files.The web server is the entry point for client requests, such as when a user accesses a web page through their browser.
It can also handle initial request processing, authentication.

When a client sends an HTTP request for a web page, the request is initially received by the web server.The web server determines whether the request is for static content (e.g., an HTML page, CSS file) or dynamic content (e.g., the result of a database query or application-specific processing).

If the request is for static content, the web server can handle it directly by serving the requested files to the client. The application server processes the request by executing server-side code, interacting with databases, and performing the necessary business logic to generate dynamic content. Once the application server has processed the request and generated the dynamic content, it sends the response back to the web server. The web server, in turn, transmits the response to the client, ensuring that the client receives the dynamic content generated by the

application server. The web server and application server work together to efficiently serve web content.

An application server is responsible for executing the application logic and dynamic content generation. It processes business logic and, interacts with databases. It is the part of the architecture where the core of the web application is implemented.

The relationship between external payment processors and a system represents how a system interacts with and relies on external payment processing services to handle financial transactions.

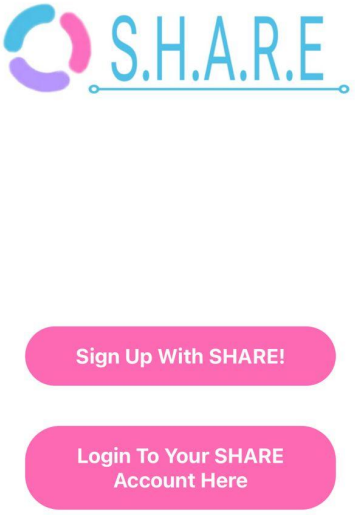
The database server is responsible for tasks such as data storage, retrieval, updating, and transaction management.

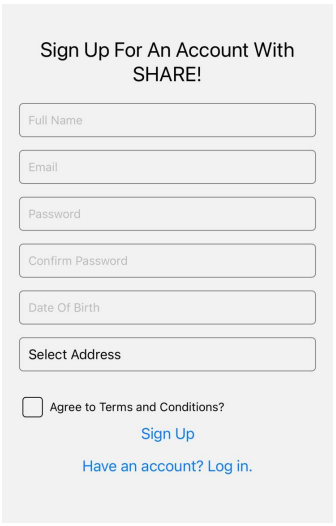
The application server hosts the application's logic and business processes. It is responsible for processing user requests, executing application-specific code, and interacting with the database server to retrieve or manipulate data.
The application server may generate dynamic content, enforce business rules, perform user authentication, and orchestrate various components of the application.

4. Software implementation

4.1. Frontend

The frontend was implemented by team member Tatiana Azar.
For the front end, each “screen” or page has its own designated class. The App.js class has a stack navigator which contains all of the screens in the project, so that they can be loaded to the app and appear there.





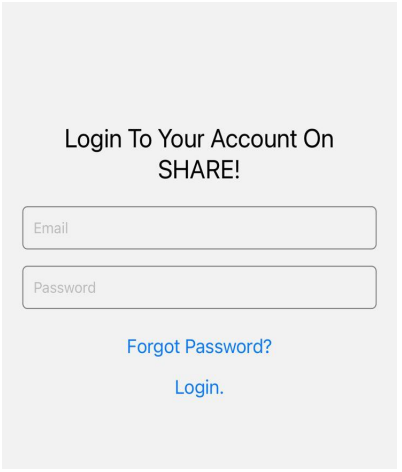


Figure 4.1.1 Opening Page Screen

Figure 4.1.2 Sign Up Screen

Figure 4.1.3 Login

For the front-end, first, there is the opening page which contains the SHARE logo and navigates the user to either a sign up screen, or the login screen. On the LoginScreen, there is a “Forgot Password?” that navigates the user to a ForgotPasswordScreen where they can enter their email and it will send them a one-time password.

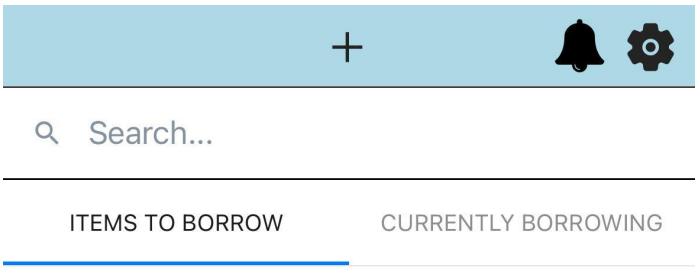
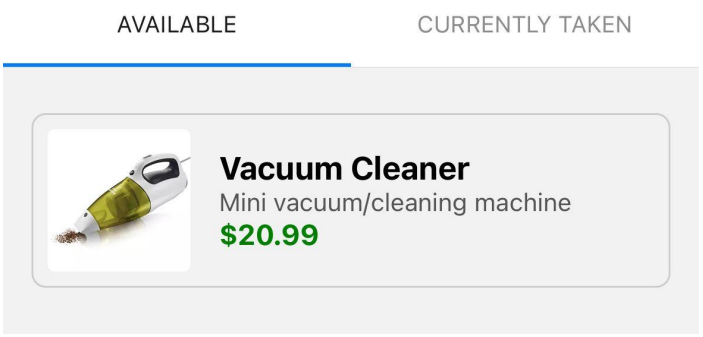


Figure 4.1.4 HomeScreen

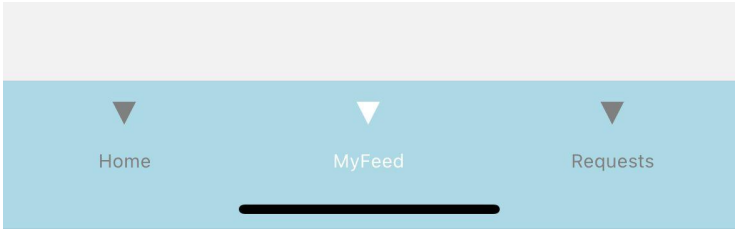
On the HomeScreen, there are 3 icons, a plus sign to allow the user to fill out a form to list an item they are willing to let other users borrow, this is on the “AddItemsToListScreen”. Once this form is sent, it adds the item along with its details to the “AvailableScreen” below. This screen is where users can view the items they have listed on the app. There is also a notifications bell, which allows users to view their notifications. With this, there is a settings icon where users can navigate to the SettingsScreen. There is the “ItemsAvailableScreen”

and the “CurrentlyBorrowingScreen” where items are displayed for users to borrow, and where items the user is currently borrowing from other users are displayed, respectively. Lastly, there is a search bar where users can search for specific items on the app they wish to borrow.



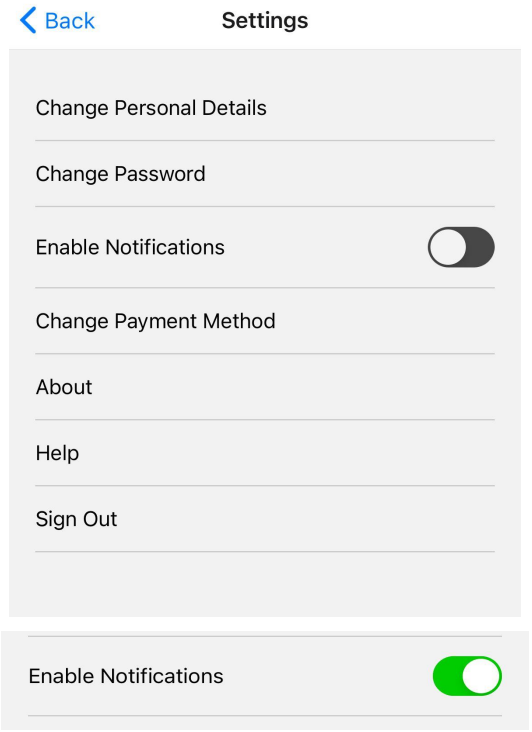
4.1.5 AvailableScreen on MyFeedScreen

Here is the AvailableScreen, with a sample item indicating the styling, rending and display of the items on the website. It includes an item image, an item name, an item description, and an item price which the borrowers will pay if they damage, break, or lose the item. The “AvailableScreen” and the “CurrentlyTakenScreen” can be navigated to from the “MyFeed” tab.



4.1.6 Bottom Bar Navigation To Navigate Between Pages

These are the bottom bar tabs which are all included inside the “TabScreen”.



4.1.7 SettingsScreen with all user setting options

The above image shows the “SettingsScreen” with various options including the “ChangePersonalDetailsScreen”, “ChangePasswordScreen”, “ChangePaymentMethodScreen”, and “SignOuScreen”. There is also a toggle switch which allows users to enable or disable notifications.

There is also a “FirstTimePaymentScreen” which is navigated to the first time a user requests to borrow an item. There is also a “NotificationsScreen”, which is navigated to when the user presses the bell icon, which displays the notifications for the user.

4.2. Backend

4.2.1 User Service

The user service project was done by team member Gazzo Mohamad and it contains the services related to the user as shown in Figure 2.1-1.

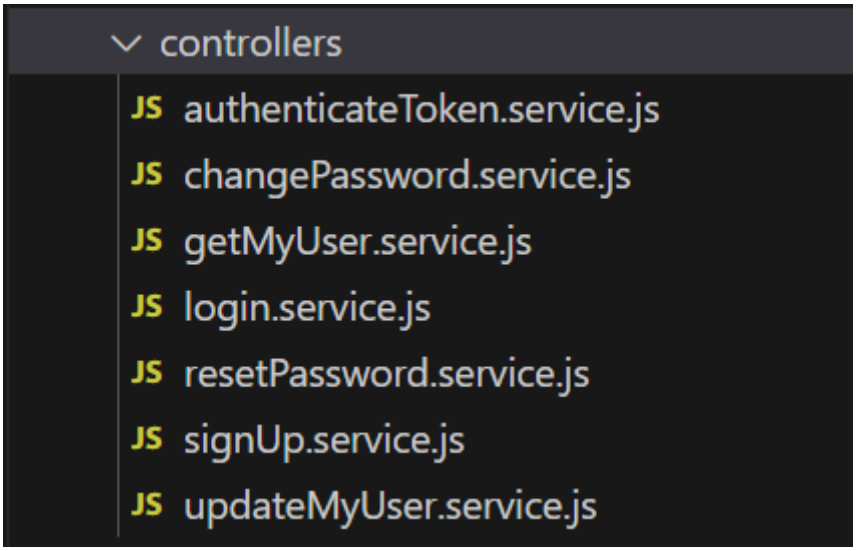


Figure 4.2.1-1 Services related to user

The authenticateToken service is responsible for ensuring the authentication token is valid. It also retrieves the user's info when they are logged in and gets deleted from the front end when the user logs out. Similarly, the getMyUser service is responsible for getting the user's information from the database.

The changePassword service is responsible for allowing the user to change their password. The service gets the user's data from Firebase using their ID and checks if the user provided the correct current password, if they did it updates the current password to the new provided password.

The login service is responsible for checking the user's username and password and if they exist in the database, it allows the user to log in.

The resetPassword service is responsible for creating a one-time password for the user when they forget their password. It finds the user's email in the database and hashes a temporary password using bcrypt. The current password in the database is updated with the temporary password so the user can use it to sign up.

The registerUser service is responsible for user registration. It validity the user's data and checks the coordinates from the location to only allow users in Leipzig, Germany to sign up for an account. It also checks if the email is not already used, and if the user name is available. If all the conditions are met, the service hashes the password and creates an entry for the new user in the database.

The updateMyUser service is responsible for updating the users profile information. It updates the users data in the database.

4.2.2 Payment Service

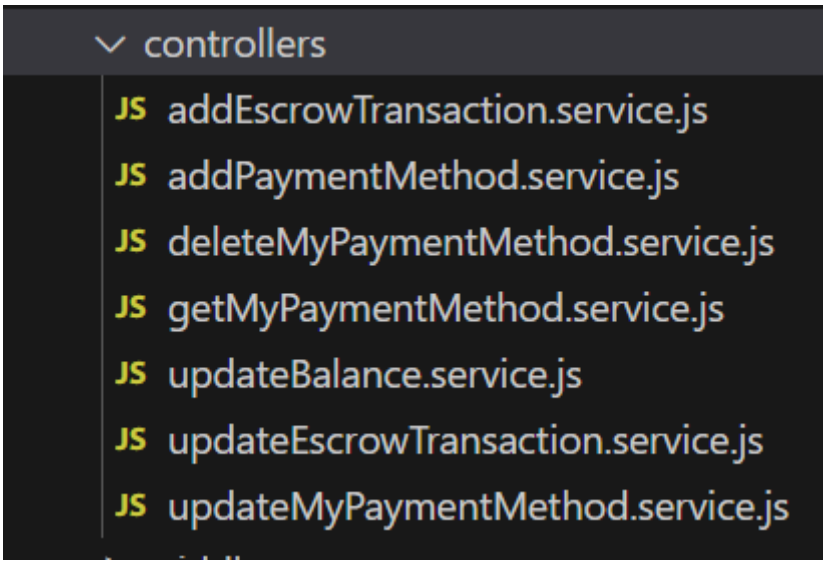


Figure 4.2.2-1 Services related to payment

The payment service project was done by the team member Gazzo Mohamad and it contains the services related to the payment. We were not able to use real payment methods in this project due to security issues so instead we opted to create a fake account for the users with a pre-determined credit amount.

The addPaymentMethod service is responsible for adding a payment method. It validates the input (credit card number, CVV, expiry date and credit card holder´s name). It also checks if the user already has a card on file before adding the new card information. When the payment method is created a default balance is assigned to the account as we can see in Figure 2.2-2.

```
res.status(httpStatus.CREATED).json({
  message: 'Payment method created successfully',
  paymentMethod: {
    id: newPaymentMethodRef.id,
    userId: req.user.id,
    cardNumber,
    cardHolderName,
    expirationDate,
    balance: parseInt(process.env.DEFAULT_BALANCE),
  },
});
```

Figure 4.2.2-2 Successful creation of a payment method

Another service in this project is the addEscrowTransaction service. This service is responsible for creating a new transaction and it does so by validating the data, updating the account balance and adding the transaction to Firebase as it is shown in Figure 2.2-3.

```
const newEscrowTransactionRef = await db
  .collection(process.env.ESCROW_TRANSACTIONS_DOC)
  .add({
    requestId,
    lenderId,
    borrowerId,
    amount,
    status: process.env.ESCROW_HELD_STATUS,
  });
```

Figure 4.2.2-3 Adding a new transaction to the database

The deleteMyPayment service is responsible for deleting the user's payment method from the database. It gets the payment information based on the user's ID and if said payment information exists, the service deletes it from Firebase.

The updateBalence service is responsible for updating the balance. It archives this by finding the balance based on the ID and adding/subtracting an amount from that balance. If the user is trying to borrow an item but they do not have enough funds in their account, the service will return an error and the user will not be able to borrow the item.

The updateEscrowTransaction service is used to either refund or transfer the amount on hold after a transaction. If the lender makes a claim that a certain item was not damaged, or if the borrower does not return an item, this service will transfer the held amount to the lender, otherwise, the amount on hold will be refunded to the borrower's account or balance.

The updateMyPaymentMethod is responsible for updating the payment method. When the user decides to add a different card to their account this service finds their old payment method in the database and updates it.

4.2.3 Notification Service

The notification service project was done by team member Gazzo Mohamad and it contains the services related to notification as shown in Figure 4.2.3-1.

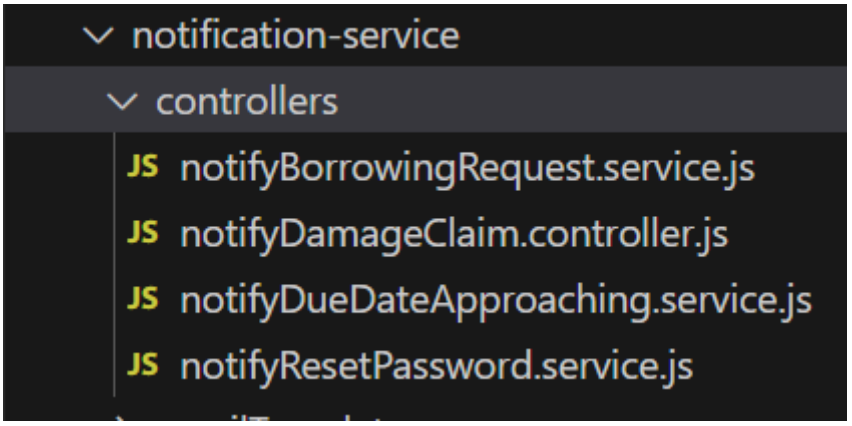


Figure 4.2.3-1 Services related to notification

The notifyBorrowingRequest service is responsible for notifying the lender when they receive a new borrowing request, the service uses WebSockets to send a real-time notification and also sends an email to the user. Similarly, the notifyDamageClaim service is responsible for notifying the borrower when the lender claims that an item they returned has been damaged. The notifyDueDateApproaching service is responsible for notifying the borrower before the due date for an item they have borrowed. Finally, the notifyResetPassword service is responsible for sending the user an email with a one-time password when the user forgets their password.

4.2.4 Item Service

The item service project was done by team member Gazzo Mohamad and it contains the services related to the item as shown in Figure 4.2.4-1.

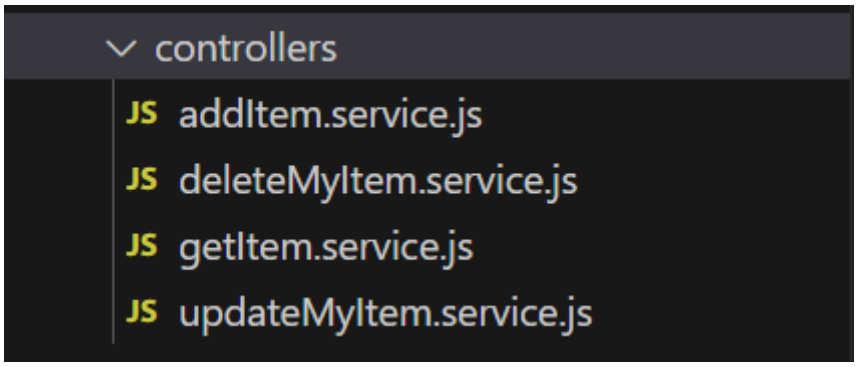


Figure 4.2.4-1

The addItem service is responsible for allowing the user to add items to the application. It takes the item name, description, price and photo and uploads it to the database.

The deleteMyItem service is responsible for allowing the user to delete a previous item they created.

Also, the getItem service is responsible for retrieving an item from the database.

Finally, the updateMyItem service is responsible for updating the information for a previously listed item.

For all the previously mentioned services, the database used was Firebase. The configuration files were set up so the services are not connected to one specific firebase project but can be connected to any project and will create the desired tables there instantly, this was done since different team members were working on different parts of the project, and so in the integration phase we are able to connect them to the same project.

Further, all the endpoints in the services were separated and tested using Postman as shown in Figures 2.4-2 and 2.4-3.

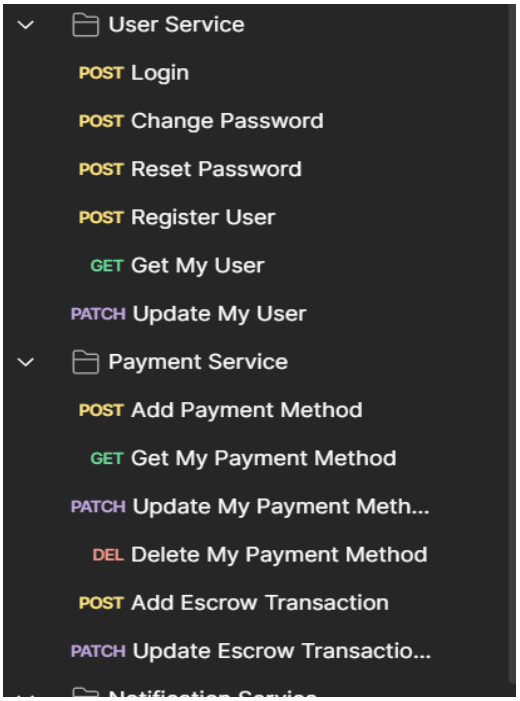


Figure 4.2.4-2 endpoints

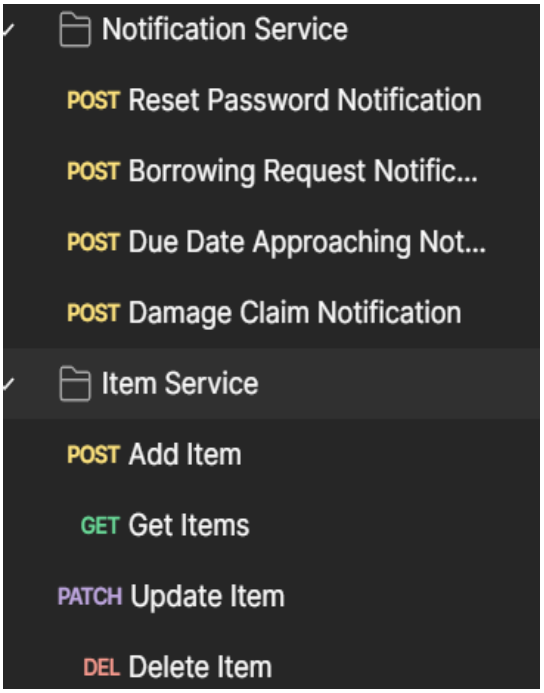


Figure 4.2.4-3 endpoints

4.2.5 Borrowing Service

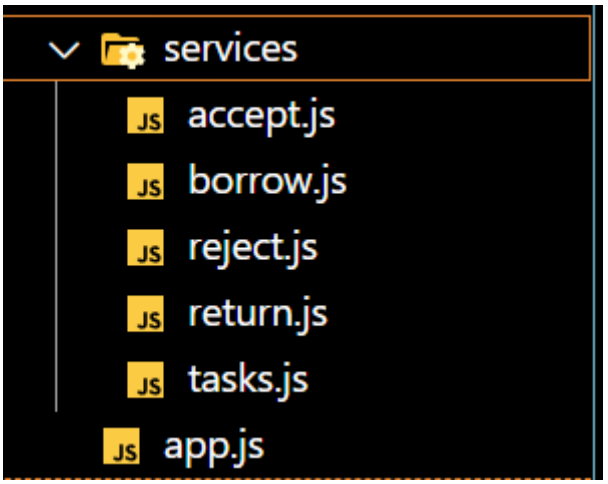


Figure 4.2.5.1

Accept.js:

- This function handles the acceptance of a borrowing request.
- It retrieves the item's information from the Firebase Realtime Database.
- If the item is in the "Requested" status, it updates the status to "Borrowed".
- It sends a JSON response with the result of the operation, including the updated item information.

Borrow.js:

- This function handles the borrowing of an item.
- It checks if the item is available (`Status` is "Available").
- If available, it updates the status to "Requested" and sends a JSON response indicating success. If not available, it informs the user that the item is not available.

Reject.js:

- This function handles the rejection of a borrowing request.
- It checks if the item is in "Requested" status, and if so, updates the status to "Available".
- Sends a JSON response indicating the result of the operation.

4.2.6 Second Borrowing Service

A different borrowing service was implemented by team member Gazzo Mohamad that contains damage claims and contains the necessary endpoints to be integrated with the front end. As shown in figure 4.2.6.1 this service contains four services: addBorrowRequest.service, getMyBorrowedItem.service, getmyLentItems.service, and updateBorrowRequest.service.

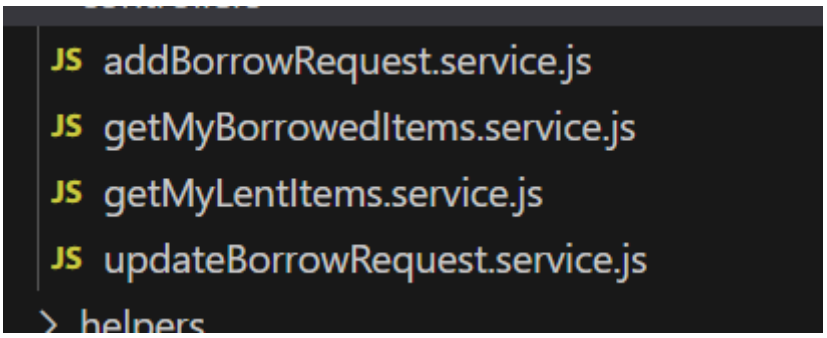


Figure 4.2.6.1 Borrowing service

The addBorrowRequest service is responsible for sending a borrow request it sends this borrowing requesti to the appropriate notification service, and the getMyBorrowedItems is responsible for showing the users the items they are currently borrowing. The getMyLentItems shows the user the items that they are currently lending to someone. The updateBorrowRequest service is responsible for updating the status of the borrowing request, figure 4.2.6.2 shows the different status of the borrow request.

```
BORROW_REQUEST_SENT_STATUS=Request Sent
BORROW_REQUEST_ACTIVE_STATUS=Active
BORROW_REQUEST_REJECTED_STATUS=Rejected
BORROW_REQUEST_RETURNED_STATUS=Returned
BORROW_REQUEST_DAMAGE_CLAIM_STATUS=Damage Claim
BORROW_REQUEST_COMPLETED_STATUS=Completed
```

Figure 4.2.6.2 Different updates for the borrowing request

5. Testing

5.1 Unit Testing

We made 4 different unit tests to test specific functions in our project. The 4 unit tests are for Sign Up, Login, Reset Password, and Change Password. All 4 of the tests are created and run the tests using the jest.mock function in javascript to create a mock database to run the tests on.

5.1.1 Sign Up Unit Test

```
describe('registerUser', () => {
  let req;
  let res;

  beforeEach(() => {
    req = {
      body: {
        name: 'John Doe',
        email: 'john@example.com',
        dob: '1990-01-01',
        latitude: 51.3397,
        longitude: 12.3731,
        password: 'password123',
      },
    };

    res = {
      status: jest.fn(() => res),
      header: jest.fn(() => res),
      json: jest.fn(() => res),
    };
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('registers a user successfully', async () => {
    bcrypt.genSalt.mockResolvedValue('mockSalt');
    bcrypt.hash.mockResolvedValue('mockHash');

    FieldValue.serverTimestamp.mockReturnValue('mockTimestamp');

    await registerUser(req, res);
```

The sign up test uses the register user API, and initialises req and res objects. The function sends a request body with the given “req” template. bycrypt.genSalt and bycrypt.hash are mocked to resolve ‘mockSalt’ and ‘mockHash’ respectively. The registerUser function is called with the req and res objects. Then, the req value is checked and a response is sent back, with a status code, header, and json, and if the request was successful, it returns a message that the user was registered successfully. After the users are added, there is a jest.clearAllMocks() function to clear the users registered from the test in the mock database. The test ensures that the response status is 201 (httpStatus.CREATED), indicating that the user was successfully created. It looks for a 'x-auth-token' in the response header. This is most likely an authentication token, suggesting that the user registration process also includes the generation of an auth token. The success message and the user's information (except the password for security reasons) are anticipated in the response JSON.

5.1.2 Login Unit Test

```
29  describe('Login function', () => {
30    it('should login successfully with valid credentials', async () => {
31      const req = {
32        body: {
33          email: 'test@example.com',
34          password: 'userPassword',
35        },
36      };
37      const res = {
38        status: jest.fn().mockReturnThis(),
39        header: jest.fn().mockReturnThis(),
40        json: jest.fn(),
41      };
42      const compareMock = jest.fn().mockResolvedValue(true);
43      bcrypt.compare.mockImplementation(compareMock);
44
45      await login(req, res);
46
47      expect(res.status).toHaveBeenCalledWith(httpStatus.OK);
48      expect(res.header).toHaveBeenCalled();
49      expect(res.json).toHaveBeenCalledWith({ message: 'Login Successful!' });
50    });
51  });
52
```

The login function uses the login API, and a request body with an email and password, which for the purpose of the test is test email and password. A response body is sent back, and the status, header and json functions are mocked to allow for verification of the response sent by the login function. The bycrypt.compare function compares the supplied password to the hashed password saved in the database. It is mocked by returning true every time, mimicking a successful password match. The test compares the mock implemented value to the mock resolved value, and ensures that the return status is 200 (httpStatus.OK), indicating that the login was successful.
The success message, Login Successful!, is anticipated in the response JSON and the test determines whether or not this message is returned.

v

5.1.3. Reset Password

```
describe('resetPassword function', () => {
  let req;
  let res;

  beforeEach(() => {
    req = {
      body: {
        email: 'johndoe@example.com',
      },
    };
    res = {
      status: jest.fn(() => res),
      json: jest.fn(),
    };
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('should reset password and send notification successfully', async () => {
    bcrypt.genSalt = jest.fn().mockResolvedValue('mockedSalt');
    bcrypt.hash = jest.fn().mockResolvedValue('mockedHash');

    const now = 'mockedTimestamp';
    generator.generate = jest.fn(() => 'mockedTemporaryPassword');

    const expectedResponse = {
      message: 'Reset password email sent successfully',
    };

    const expectedUpdate = {
      password: null,
      temporary_password: 'mockedHash',
      modified_at: now,
    };

    const originalServerTimestamp = require('../../startup/firebase').FieldValue.serverTimestamp;
    require('../../startup/firebase').FieldValue.serverTimestamp = jest.fn(() => now);

    await resetPassword(req, res);

    expect(res.status).toHaveBeenCalledWith(httpStatus.OK);
    expect(res.json).toHaveBeenCalledWith(expectedResponse);
  });
```

The reset password function has a request and response, and it initialises them with mock data and functions. After each test, this information is cleared with the jest.clearAllMocks() function. The test case is using the bycrypt library to hash the passwords, and the bycrypt.genSalt and the bycrypt.hash test mocks return predefined values. This avoids the process of actually computing password hashes during the tests to make it faster. The test sets up a mocked timestamp and temporary password. There is the requested response which returns the message that the reset password email was successfully sent.The test calls the function with the mock request and response, and checks if the function behaves as intended, and if yes, it returns a status code of ‘OK’ with the message given if the test is

successful. The test lastly checks if the JSON content and the status code response are equal to the expected response outcomes.

5.1.4. Change Password Unit Test

```
describe('changePassword function', () => {
  it('should change the password successfully', async () => {
    const req = {
      body: {
        currentPassword: 'oldPassword',
        newPassword: 'newPassword',
      },
      user: {
        id: 'mockedUserId',
      },
    };
    const res = {
      status: jest.fn(() => res),
      json: jest.fn(),
    };

    bcrypt.compare = jest.fn(() => true);
    bcrypt.genSalt = jest.fn(() => 'mockedSalt');
    bcrypt.hash = jest.fn(() => 'mockedHashedPassword');

    await changePassword(req, res);

    expect(res.status).toHaveBeenCalledWith(httpStatus.OK);
    expect(res.json).toHaveBeenCalledWith({ message: 'Password changed successfully' });
  });
});

```

The changePassword function has req, being a request object which simulates an HTTP request body containing the currentPassword and the newPassword, as well as the user id. There is also the response object, res, creates a mosque status and json function to allow for verification that the response was sent. When the function is executed under the test the changePassword function is called with the mock values for the req and res. The function changes the user's password if the currentPassword matches the user's existing password. The response status is checked to verify that it returns a status code 'OK' and that the response JSON contains the message "Password changed successfully".

5.2 Integration testing

5.2.1 Sign-up and login test

This is an automated test to test the integration of the sign-up service, and login service with the database. It uses the jest mock function in node js and works by signing up and creating an account for a mock user and then logging in to this user account.

The function in Figure 5.2.1.1 is responsible for deleting this mock user after the test is completed, this is done to ensure the database is not flooded with mock users from running the test.

```
jest.mock('../../helpers/location', () => jest.fn());

afterAll(async () => {
  await testDb
    .collection(process.env.USERS_DOC)
    .get()
    .then((snapshot) => {
      snapshot.docs.forEach((doc) => {
        testDb.collection(process.env.USERS_DOC).doc(doc.id).delete();
      });
    });
});
```

Figure 5.2.1.1 - Deleting users after testing

Then a user is created as shown in figure 5.2.1.2 and after the registration is successful as shown in figure 5.2.1.3. After, the test logs in with the mock user as shown in figure 5.2.1.4.

```
it('should signup a user and then login successfully', async () => {
  const signupReq = {
    body: {
      name: 'Test User',
      email: 'test@example.com',
      dob: '1990-01-01',
      latitude: 51.3397,
      longitude: 12.3731,
      password: 'testpassword',
    },
```

Figure 5.2.1.2 Creating a mock user

```
});

await registerUser(signupReq, signupRes, testDb, FieldValue);

expect(signupRes.status).toHaveBeenCalledWith(httpStatus.CREATED);
expect(signupRes.json).toHaveBeenCalledWith({
  message: 'User registered successfully',
  user: {
```

Figure 5.2.1.3 receiving a success HTTP status for creating the user

```
const loginRes = {
  status: jest.fn().mockReturnThis(),
  header: jest.fn().mockReturnThis(),
  json: jest.fn(),
};
```

Figure 5.2.1.4 Logging in

The test is successful if it can sign up and log in successfully. Some fail conditions are if the password that is used for login is different than the one for sign up, or if the location used during the sign up is not in Leipzig Germany.

5.2.2 Add items and update items test

This is an automated test to test the integration of the add items service, update items test and the database. It adds an item to the database and then retrieves that item using the id and updates it.

Similar to the sign-up and log-in test, it has a function that deletes everything added to the database after the test is done as shown in Figure 5.2.2.1. An item is added to the database with a known id as shown in Figure 5.2.2.2. Then this item is retrieved and the item information is updated as shown in Figure 5.2.2.3.

```
afterAll(async () => {
  await testDb.collection(process.env.ITEMS_DOC).doc(createdItemId).delete();
});
```

Figure 5.2.2.1 Deleting anything added to the database after the test is done

```
const addItemReq = {
  body: {
    itemName: 'Test Item',
    itemDescription: 'Initial description',
    price: '100',
  },
  user: {
    id: 'testUserId',
  },
  files: '',
};

const addItemRes = {
  status: jest.fn(() => addItemRes),
  json: jest.fn(),
};
```

Figure 5.2.2.2 Adding an item to the database

```
createdItemId = addItemRes.json.mock.calls[0][0].item.id;

const updateItemReq = {
  params: {
    id: createdItemId,
  },
  body: {
    itemName: 'Updated Item',
    itemDescription: 'Updated description',
    price: '200',
    unavailabeDurations: '2024-01-01',
  },
  user: {
    id: 'testUserId',
  },
};
```

Figure 5.2.2.3 Updating an item

The test is successful if it can add an item and update it successfully. Some fail conditions are if the ID used to retrieve the item for updating does not match the ID of the created item or if the connection path to the database is wrong.

5.3 Regression testing

5.3.1 Change password with minimum password length

In order to make the app more secure, the change password service was changed to force the user to have a password longer than 10 characters, so to test the new service. They new unit test will only be successful if the password is longer than 10 characters as shown in Figure 5.3.1.1.

```
const req = {
  body: {
    currentPassword: 'Password',
    newPassword: 'newPassword',
  },
  user: {
    id: 'mockedUserId',
  },
};
const res = {
  status: jest.fn(() => res),
  json: jest.fn(),
};

bcrypt.compare = jest.fn(() => true);
bcrypt.genSalt = jest.fn(() => 'mockedSalt');
bcrypt.hash = jest.fn(() => 'mockedHashedPassword');
```

Figure 5.3.1.1 Changing the password in improved Service

5.3.2 Log in with minimum password length

After improving the changing password service, the same improvement was added to the log-in service. A new unit test was made to test the new service. As shown in figure 5.3.2.1 the new login has to be with a password longer than 10 characters or it will give an error.

```
it('should give an error for password length validation', async () => {
  const req = {
    body: {
      email: 'test@example.com',
      password: 'user',
    },
  };
  const res = {
```

Figure 5.3.2.1 Password Length Validation