Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Canteen on the Go

Project Engineering

Year 4

Patryk Milkiewicz

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2023/2024

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 Galway-Mayo Institute of Technology.

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.

____

Acknowledgements

I would like to thank my supervisor, Niall O'Keefe, for his guidance, advice and support through the year.

I would like to thank all the other supervisors, Brian O'Shea, Michelle Lynch, Paul Lennon, Ben Kinsella, and David Newell who have shown their support throughout the year and provided good feedback to everyone.

Table of Contents

1 Summary

When I started the year, I thought about my project. I wanted to solve a real-world problem.

The idea came to me when I was buying lunch in the college canteen. During the lunch rush the canteen would get overwhelmed. The main reasons for this were the large amount of people buying lunch and the card readers which frequently experienced problems with connection when confirming payments.

I decided to create an express checkout system. This system would allow customers to quickly pay for their lunch, and it would also help to reduce queues and congestion in the canteen. It would also help to speed up the payment process, making the overall experience more efficient. My main inspiration was the Amazon Go shops, and I wanted to create something similar.

My project consists of a High-Frequency RFID Reader, RFID Tags, and a mobile app. The RFID tags would come pre-programmed on the products and would contain data such as product name, price, and product code. There is also an RFID payment card, which would get picked up by the reader along with the products, allowing for ultra-fast checkout. The mobile app allows students to top up their accounts using debit cards and check their purchase history.

For the hardware side, I chose the M5Stack. The M5Stack is a leading provider of IoT solutions, offering stackable hardware modules featuring ESP32 boards [1]. Their M5Atom Matrix offers an LED display and the ESP32PICO and I have used this module to connect to their M5Stack UHF-RFID JRD-4035 Module

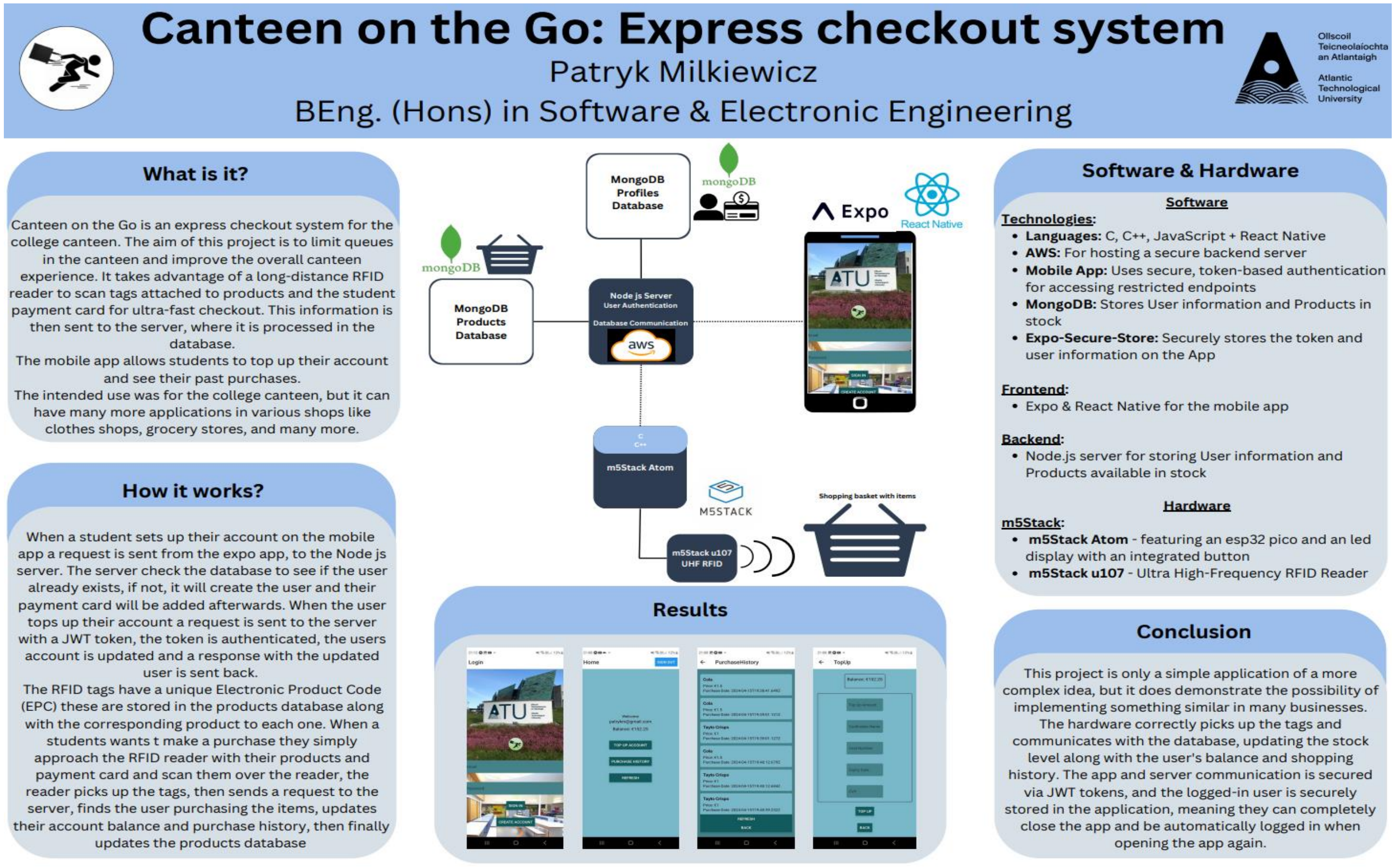
On the software side, I created a mobile app using JavaScript and React Native. This app securely communicates with a Node.js server hosted on AWS cloud infrastructure, utilizing token authentication for enhanced security.

In the end, I managed to correctly identify multiple tags being scanned at once and have those tags sent to my server where they would be processed, updating the product database and the

user database. The mobile app securely connects to the server and allows the user to top up their account and view their past purchases.

While the project isn't without its imperfections, it lays a solid foundation for future development. By optimizing checkout processes and enhancing user experience, my solution not only addresses immediate challenges but also sets the stage for ongoing improvements in the canteen operations.

2 Poster



Canteen on the Go: Express checkout system

Patryk Milkiewicz

BEng. (Hons) in Software & Electronic Engineering

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

What is it?

Canteen on the Go is an express checkout system for the college canteen. The aim of this project is to limit queues in the canteen and improve the overall canteen experience. It takes advantage of a long-distance RFID reader to scan tags attached to products and the student payment card for ultra-fast checkout. This information is then sent to the server, where it is processed in the database.

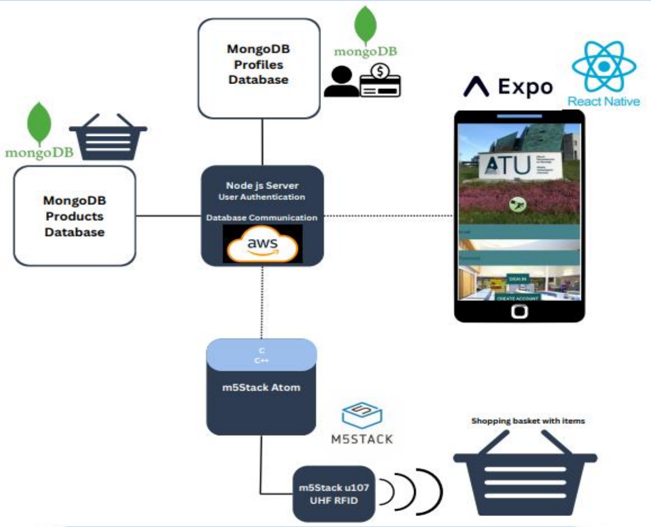
The mobile app allows students to top up their account and see their past purchases.

The intended use was for the college canteen, but it can have many more applications in various shops like clothes shops, grocery stores, and many more.

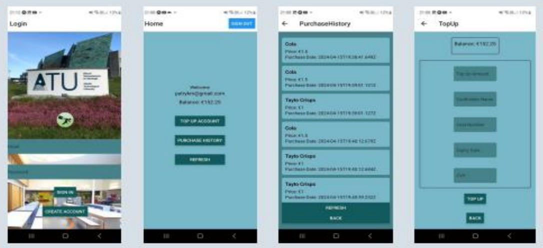
How it works?

When a student sets up their account on the mobile app a request is sent from the expo app, to the Node js server. The server check the database to see if the user already exists, if not, it will create the user and their payment card will be added afterwards. When the user tops up their account a request is sent to the server with a JWT token, the token is authenticated, the users account is updated and a response with the updated user is sent back.

The RFID tags have a unique Electronic Product Code (EPC) these are stored in the products database along with the corresponding product to each one. When a students wants t make a purchase they simply approach the RFID reader with their products and payment card and scan them over the reader, the reader picks up the tags, then sends a request to the server, finds the user purchasing the items, updates their account balance and purchase history, then finally updates the products database



Results



Software & Hardware

Software

Technologies:

- **Languages:** C, C++, JavaScript + React Native
- **AWS:** For hosting a secure backend server
- **Mobile App:** Uses secure, token-based authentication for accessing restricted endpoints
- **MongoDB:** Stores User information and Products in stock
- **Expo-Secure-Store:** Securely stores the token and user information on the App

Frontend:

- Expo & React Native for the mobile app

Backend:

- Node.js server for storing User information and Products available in stock

Hardware

m5Stack:

- **m5Stack Atom** - featuring an esp32 pico and an led display with an integrated button
- **m5Stack u107** - Ultra High-Frequency RFID Reader

Conclusion

This project is only a simple application of a more complex idea, but it does demonstrate the possibility of implementing something similar in many businesses. The hardware correctly picks up the tags and communicates with the database, updating the stock level along with the user's balance and shopping history. The app and server communication is secured via JWT tokens, and the logged-in user is securely stored in the application, meaning they can completely close the app and be automatically logged in when opening the app again.

Figure 2-1 Project Poster

3 Introduction

In today’s fast-paced world, efficiency is key, especially when it comes to daily routines like grabbing lunch. Nobody likes to be stuck in queues, and it’s even more infuriating when there is a long queue and you’re buying a single small item like coffee or crisps for a quick snack. This becomes even more annoying when the checkout payment system is very slow. Let's be honest, young people rarely carry cash anymore; after all, we live in the digital era.

It was the primary purpose of this project to develop an express checkout system for the college canteen, in response to the need for a swift and seamless transaction process, which inspired the development of this system. The system was designed to streamline the process of buying items in the canteen, allowing students to quickly purchase their items and exit the canteen. It was also intended that the improvement would reduce the length of time students would have to wait in line as well as help the canteen staff to better manage their day-to-day operations.

Within the scope of the project, my aim was to create a system which would not only facilitate fast payments, but also minimize congestion and improve the overall efficiency during busy periods. By using a combination of hardware and software technologies, my objective was to design a solution that enhances convenience for students while providing valuable insight for canteen management.

4 Background

4.1 Existing Solutions

Amazon’s ‘Just Walk Out’: Amazon is the current market leader in providing cashier-less technology to stores. It's powered by Amazon's Just Walk Out shopping. Any items taken from the shelf are added to a virtual cart and removed when put back on the shelf. When leaving the store, the payment is automatically processed through the Amazon app and a receipt is sent to your email address.

This seems very simple at first glance. However, the technology behind this system is complex and requires sophisticated computer vision, sensors, and artificial intelligence. This technology allows Amazon Go to be more efficient and cost-effective than traditional stores. However, one potential challenge of implementing this technology is the reliance on advanced computer vision and sensors. Any malfunctions or inaccuracies in these systems could lead to errors in item detection and incorrect charges. Additionally, the technology may struggle with accurately identifying items that are placed back on the shelf, resulting in charging customers for items they did not actually purchase.

[1]Recent reports have revealed that Amazon's 'Just Walk Out' technology relies significantly on human moderators and data labelers to review transactions and train AI models. With a team of over 1,000 employees, primarily based in India, about 700 out of every 1,000 Just Walk Out sales required manual review in 2022, surpassing Amazon's internal goal of 50 per 1,000 sales. This sheds light on the extent of human involvement behind the scenes, contrasting with the technology's portrayal as fully automated.

Standard Cognition: This company provides an autonomous tool that can be installed into retailers’ existing shops. This is similar technology to Amazons’ ‘Just Walk Out’, relying on cameras and AI to keep track of shoppers and products. The system uses computer vision, deep learning, and deep learning algorithms to track inventory and detect when a customer leaves the store. It also uses AI-powered analytics to predict product demand and optimize store layout.

Both solutions require large amounts of data to train the models, which is then used to optimize store operations. The system can also detect suspicious activity in real-time, helping to prevent shoplifting and theft in stores. These systems are complex and require significant computing power to process and analyze data. They also require regular maintenance to ensure accuracy.

I asked myself, "Why is this so complex and expensive?". What if I could create a system that is simpler than this but still serves the same purpose? Not everything needs to be packed with expensive technology.

It turned out most shops already have this technology in place but use it for a different purpose. It's being used as an anti-theft device. I'm talking about security gates. If these gates are so good at picking up stolen items, then why wouldn't they be able to pick up an assortment of items and replace the whole checkout system?

4.2 Tools & Technologies

In my project, I applied a range of modern technologies, drawing on my knowledge and skills from college. I developed an innovative solution to the canteen issue using the following tools and technologies:

4.2.1 Arduino IDE

[2]Arduino IDE is an open-source integrated development environment (IDE) for the Arduino platform. It is used to develop software for microcontrollers based on the Arduino platform. It is free to download and use, and it supports a variety of languages including C++, C, and Python.

It allows for the installation of external libraries, allowing me to use it to write code for the esp32 microcontroller.

4.2.2 M5Stack

4.2.2.1 M5Atom Matrix

[3]The ATOM Matrix is powered by the ESP32-PICO-D4 chip, providing Wi-Fi connectivity and 4MB of flash memory. It features an Infra-Red LED, a 5 * 5 RGB LED matrix, and an IMU sensor (MPU6886) for motion sensing. A programmable button allows for easy user input, and a Type-C USB interface facilitates fast program uploading. With its compact size, powerful features, and convenient mounting options, the ATOM Matrix is an ideal choice for developers seeking a versatile and compact solution for embedded device development.

4.2.2.2 M5Stack UHF-RFID

[4]The UHF-RFID module is an ultra-high frequency wireless reader with a built-in ceramic antenna, eliminating the need for additional antennas. It features optimized RF design for low power consumption and high performance, with a transmission power of 100mW reaching distances of over 1.5 meters. Utilizing serial communication interface and plug-and-play AT command set, it offers easy development and usage. Suitable for warehousing logistics management and smart retail, it meets requirements for monitoring and reading multiple product tags.

4.2.3 React Native

[5] React Native, an open-source framework by Meta Platforms, Inc., enables cross-platform app development for Android, iOS, and more. It leverages React framework and native capabilities, used by Facebook, Microsoft, and Shopify. Unlike React, it doesn't manipulate DOM via Virtual DOM but communicates with native platforms via a bridge. React components interact with native APIs, often using TypeScript for type safety. Styling resembles CSS but operates with native views, not HTML or CSS.

4.2.4 Expo Go

[6]Expo Go is a mobile app that allows for testing and direct preview of React Native apps. It provides a convenient way to see real-time updates to the app when changes are made to the

code. Expo Go eliminates the need for complex setup or configuration by providing a streamlined development environment. It offers features like hot reloading for instant code updates, device emulation for testing various screen sizes, and access to native device functionality through Expo's extensive library of pre-built components and APIs.

4.2.5 Express.js

[7]Express is a web application framework designed for creating RESTful APIs using Node.js. It is available as free and open-source software under the MIT License. Express serves as the backend component in well-known development stacks such as MEAN, MERN, or MEVN, working alongside MongoDB database software and a JavaScript front-end framework or library.

4.2.6 Node.js

[8]Node.js is an open-source, server-side JavaScript runtime environment that allows developers to build scalable, networked applications. It utilizes an event-driven, non-blocking I/O model, making it lightweight and efficient at handling concurrent connections. Node.js is commonly used for building web servers, APIs, and real-time applications. It provides a rich ecosystem of libraries and frameworks, including Express.js for building web applications. Node.js enables developers to use JavaScript both on the client and server-side, promoting code reuse and simplifying the development process for full-stack applications.

4.2.7 MongoDB

[9]MongoDB is a popular NoSQL database management system known for its flexibility, scalability, and performance. It stores data in flexible, JSON-like documents, making it easy to handle and manipulate data in a schema-less environment.

Mongoose

[10]Mongoose is a MongoDB and Node.js object data modelling library. It enforces data integrity and validation while simplifying MongoDB interactions with a straightforward schema-based solution. Using Mongoose, developers can define schemas and corresponding models for CRUD (Create, Read, Update, Delete) operations in MongoDB collections. In Node.js

applications, Mongoose offers features such as data casting, query building, middleware, and schema validation.

4.2.8 Amazon Web Services (EC2)

[11]Amazon Web Services (AWS) is a comprehensive cloud computing platform offered by Amazon. It provides a wide range of services, including computing power, storage, databases, machine learning, and more, delivered over the internet on a pay-as-you-go basis. Among its core services is Amazon Elastic Compute Cloud (EC2), which offers scalable virtual servers in the cloud, allowing users to quickly deploy and manage computing resources. EC2 instances can be easily configured to meet specific requirements, providing flexibility and scalability for various applications, from simple web hosting to complex enterprise workloads.

5 Project Architecture

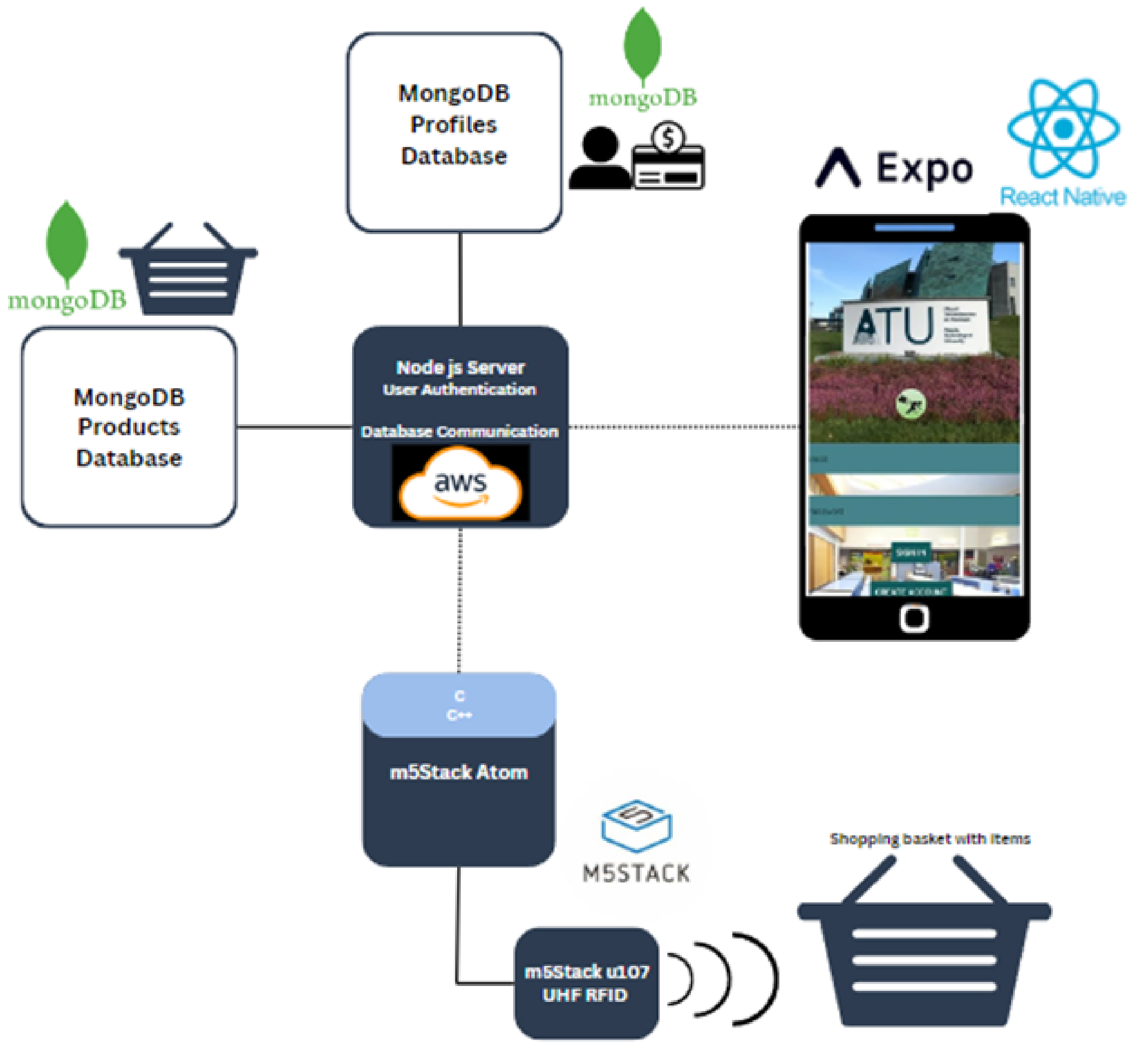


Figure 5-1 Architecture Diagram

6 Project Plan

For my project planning I have used a rough sketch:

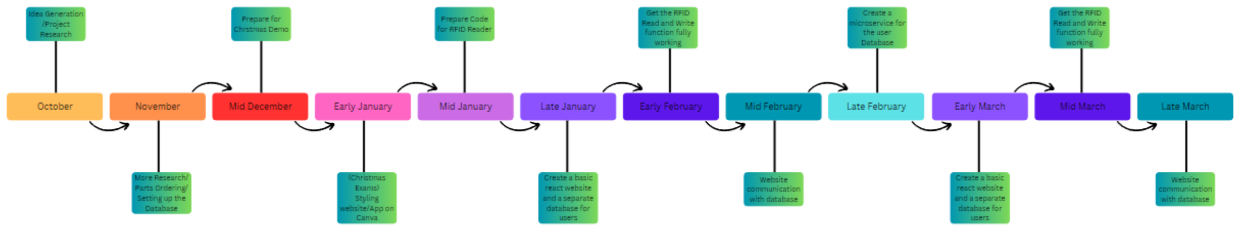
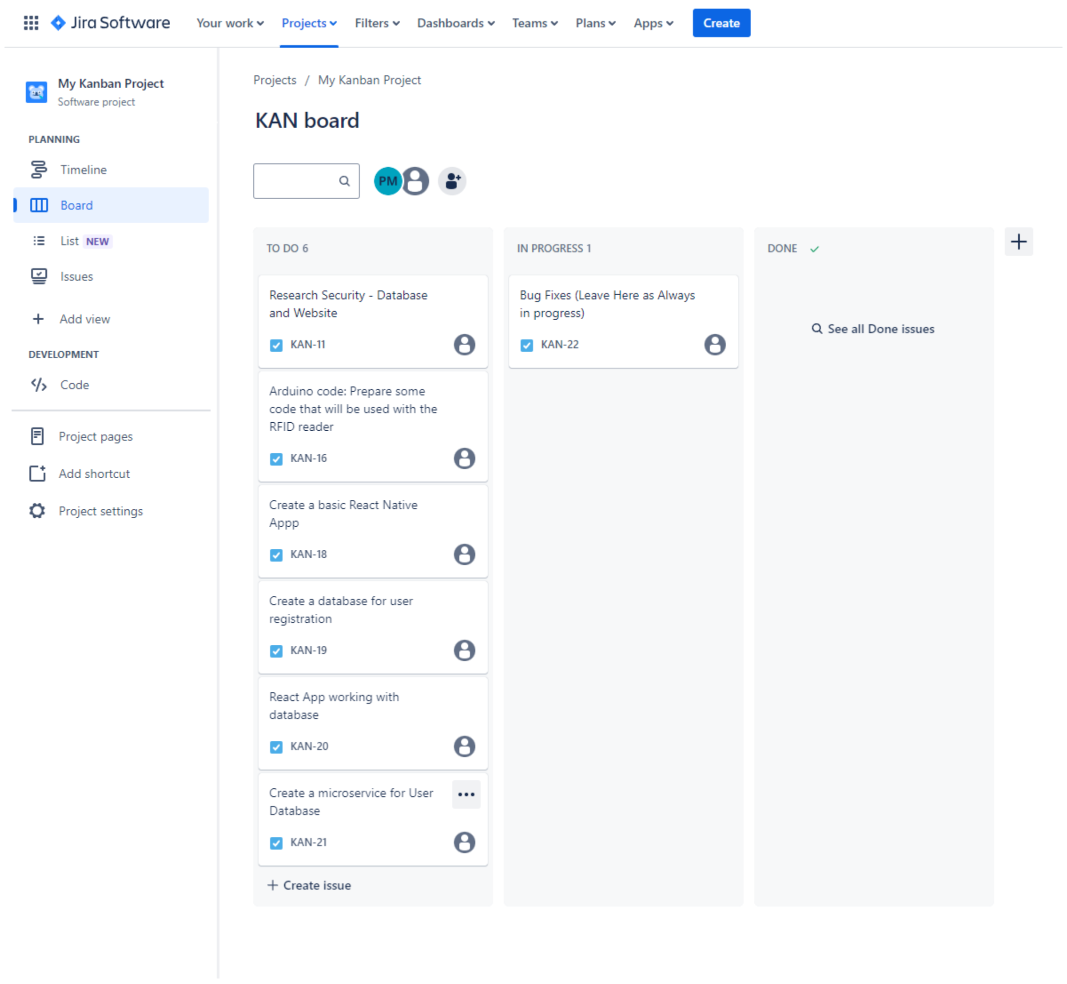

Idea Generation /Project Research
October
More Research/ Parts Ordering/ Setting up the Database
November
Prepare for Chrstmas Demo
Mid December
(Christmas Exams) Styling website/App on Canva
Early January
Prepare Code for RFID Reader
Mid January
Create a basic react website and a separate database for users
Late January
Get the RFID Read and Write function fully working
Early February
Website communication with database
Mid February
Create a microservice for the user Database
Late February
Create a basic react website and a separate database for users
Early March
Get the RFID Read and Write function fully working
Mid March
Website communication with database
Late March

Figure 6-1 project plan

And kept track of my tasks using Kanban:



KAN board

TO DO 6	IN PROGRESS 1	DONE ✓
Research Security - Database and Website KAN-11	Bug Fixes (Leave Here as Always in progress) KAN-22	See all Done issues
Arduino code: Prepare some code that will be used with the RFID reader KAN-16		
Create a basic React Native Appp KAN-18		
Create a database for user registration KAN-19		
React App working with database KAN-20		
Create a microservice for User Database KAN-21		
+ Create issue		

Figure 6-2 Kanban

7 Hardware Connection

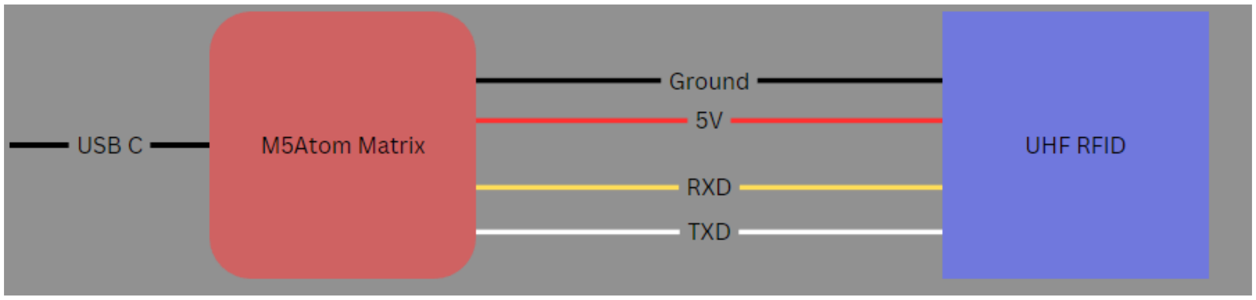


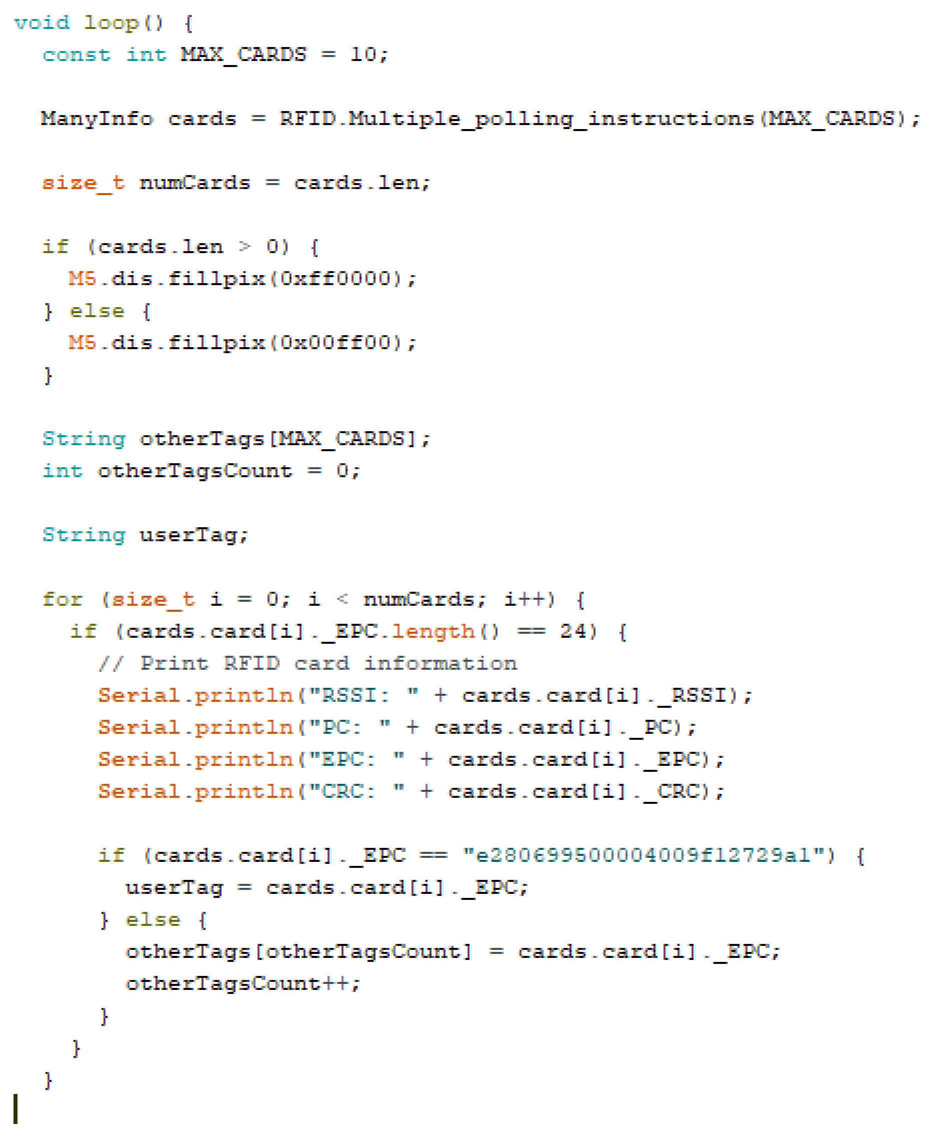
Figure 7-1 Microcontroller & RFID Module Connection

The UHF RFID Reader/Writer is connected directly to the M5Atom Matrix by using a ribbon cable included with the RFID Module. There is no need for logic converters or extra resistors thanks to the M5Stack architecture. It's a simple plug-and-play solution, which makes it quick and easy to integrate the RFID reader into any M5Stack project.

The M5Atom connects directly to a power source/computer using a USB C cable.

8 Project Code

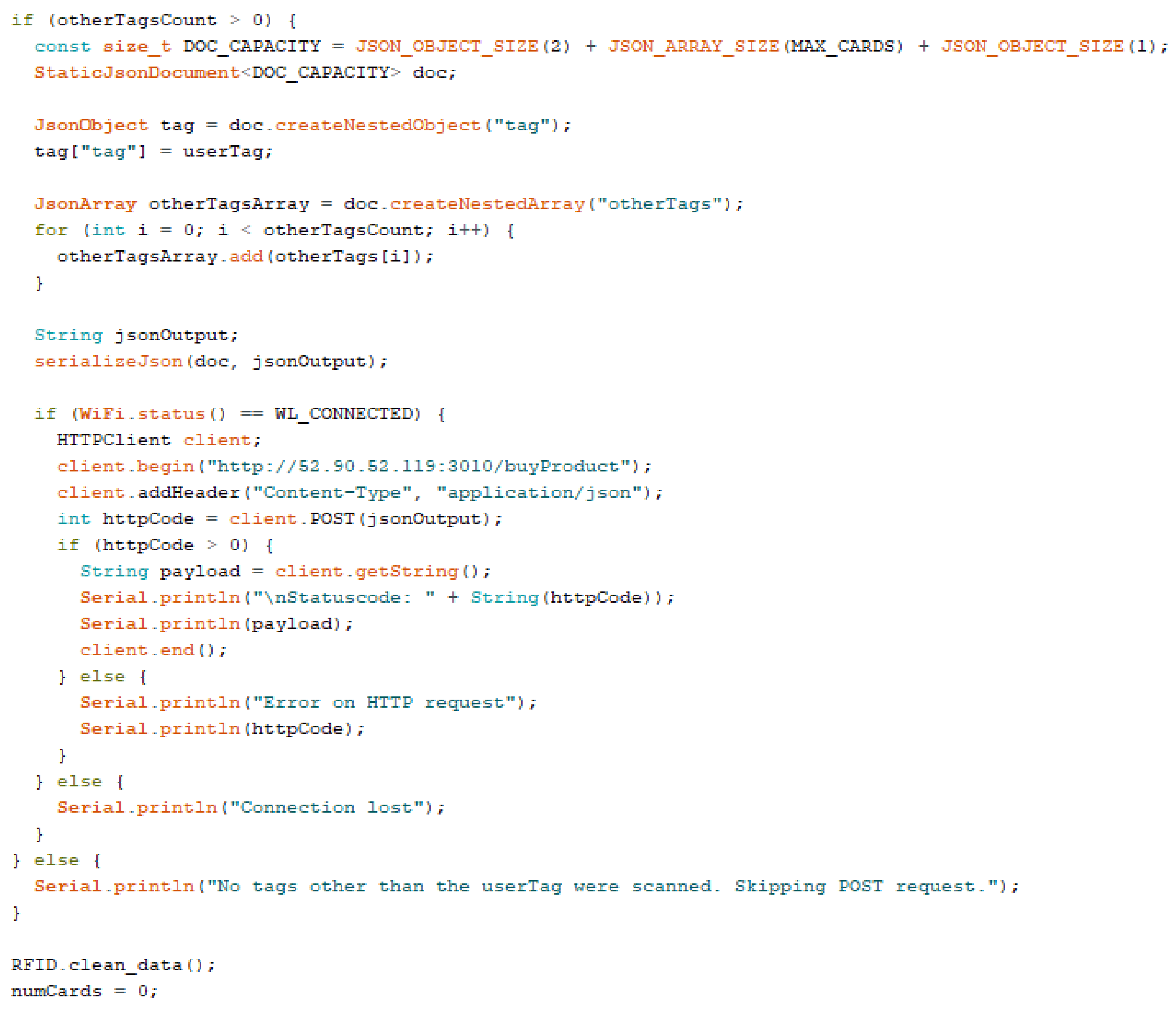
8.1 UHF RFID



```
void loop() {
  const int MAX_CARDS = 10;

  ManyInfo cards = RFID.Multiple_polling_instructions(MAX_CARDS);

  size_t numCards = cards.len;

  if (cards.len > 0) {
    M5.dis.fillpix(0xff0000);
  } else {
    M5.dis.fillpix(0x00ff00);
  }

  String otherTags[MAX_CARDS];
  int otherTagsCount = 0;

  String userTag;

  for (size_t i = 0; i < numCards; i++) {
    if (cards.card[i]._EPC.length() == 24) {
      // Print RFID card information
      Serial.println("RSSI: " + cards.card[i]._RSSI);
      Serial.println("PC: " + cards.card[i]._PC);
      Serial.println("EPC: " + cards.card[i]._EPC);
      Serial.println("CRC: " + cards.card[i]._CRC);

      if (cards.card[i]._EPC == "e280699500004009f12729a1") {
        userTag = cards.card[i]._EPC;
      } else {
        otherTags[otherTagsCount] = cards.card[i]._EPC;
        otherTagsCount++;
      }
    }
  }
```

Figure 8-1 Reading tags

In my main Arduino loop I call the already defined function called ‘**Multiple_polling_instructions**’ which scans multiple tags and stores them temporarily. These tags are then returned from the function, and I extract the EPC code of each tag and store them in an array. The user tag here is pre-defined and acts as a payment card.



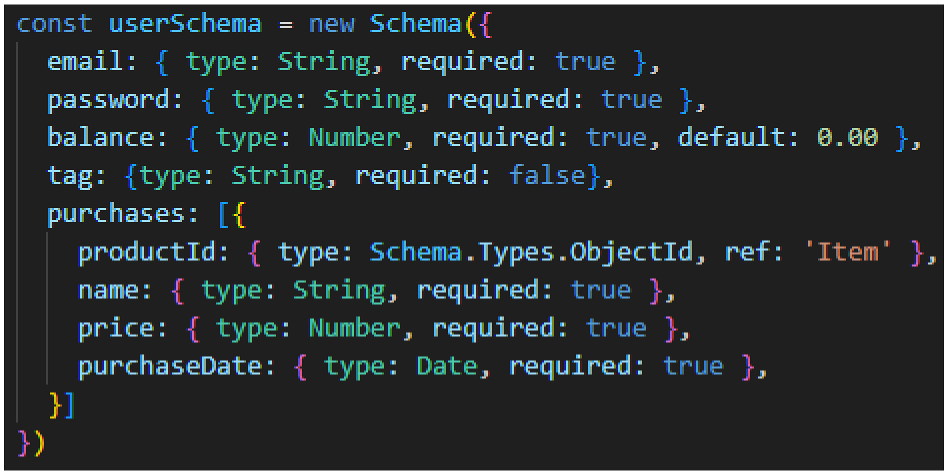
```
if (otherTagsCount > 0) {
  const size_t DOC_CAPACITY = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(MAX_CARDS) + JSON_OBJECT_SIZE(1);
  StaticJsonDocument<DOC_CAPACITY> doc;

  JsonObject tag = doc.createNestedObject("tag");
  tag["tag"] = userTag;

  JsonArray otherTagsArray = doc.createNestedArray("otherTags");
  for (int i = 0; i < otherTagsCount; i++) {
    otherTagsArray.add(otherTags[i]);
  }

  String jsonOutput;
  serializeJson(doc, jsonOutput);

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient client;
    client.begin("http://52.90.52.119:3010/buyProduct");
    client.addHeader("Content-Type", "application/json");
    int httpCode = client.POST(jsonOutput);
    if (httpCode > 0) {
      String payload = client.getString();
      Serial.println("\nStatuscode: " + String(httpCode));
      Serial.println(payload);
      client.end();
    } else {
      Serial.println("Error on HTTP request");
      Serial.println(httpCode);
    }
  } else {
    Serial.println("Connection lost");
  }
} else {
  Serial.println("No tags other than the userTag were scanned. Skipping POST request.");
}

RFID.clean_data();
numCards = 0;
```

Figure 8-2 Sending tags to server

In this part of the code, firstly the code calculates the capacity required for the JSON document (‘DOC_CAPACITY’) based on the number of tags. Then I create a ‘**StaticJsonDocument’** with this capacity. I then create a JSON object with the ‘**userTag’** and a JSON array with the product tags. This JSON document is then serialized into a string and a POST request is sent to my server with all this information in the header so it can be further processed there.

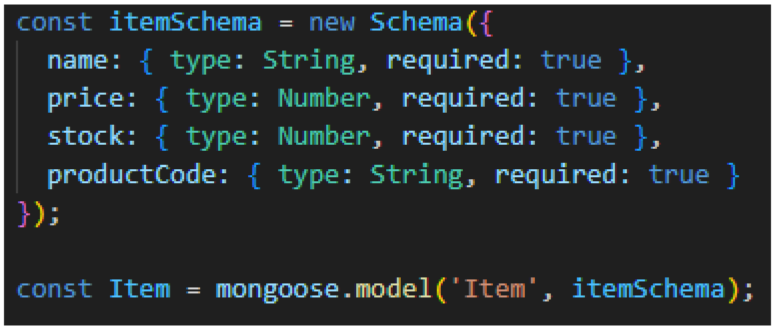
8.2 Back end

A back end is the part of a software system or application that operates behind the scenes, handling data processing, database management, and server-side logic. It serves as the backbone of the application, supporting and enabling its functionality. In the context of my project, I'm using the back end to manage user authentication, handle database interactions, and process requests from the front end. Essentially, the back end handles the 'behind-the-scenes' tasks, allowing the front end to interact with users and display information in a user-friendly manner.



```
const userSchema = new Schema({
  email: { type: String, required: true },
  password: { type: String, required: true },
  balance: { type: Number, required: true, default: 0.00 },
  tag: {type: String, required: false},
  purchases: [{
    productId: { type: Schema.Types.ObjectId, ref: 'Item' },
    name: { type: String, required: true },
    price: { type: Number, required: true },
    purchaseDate: { type: Date, required: true },
  }]
})
```

Figure 8-3 User Schema

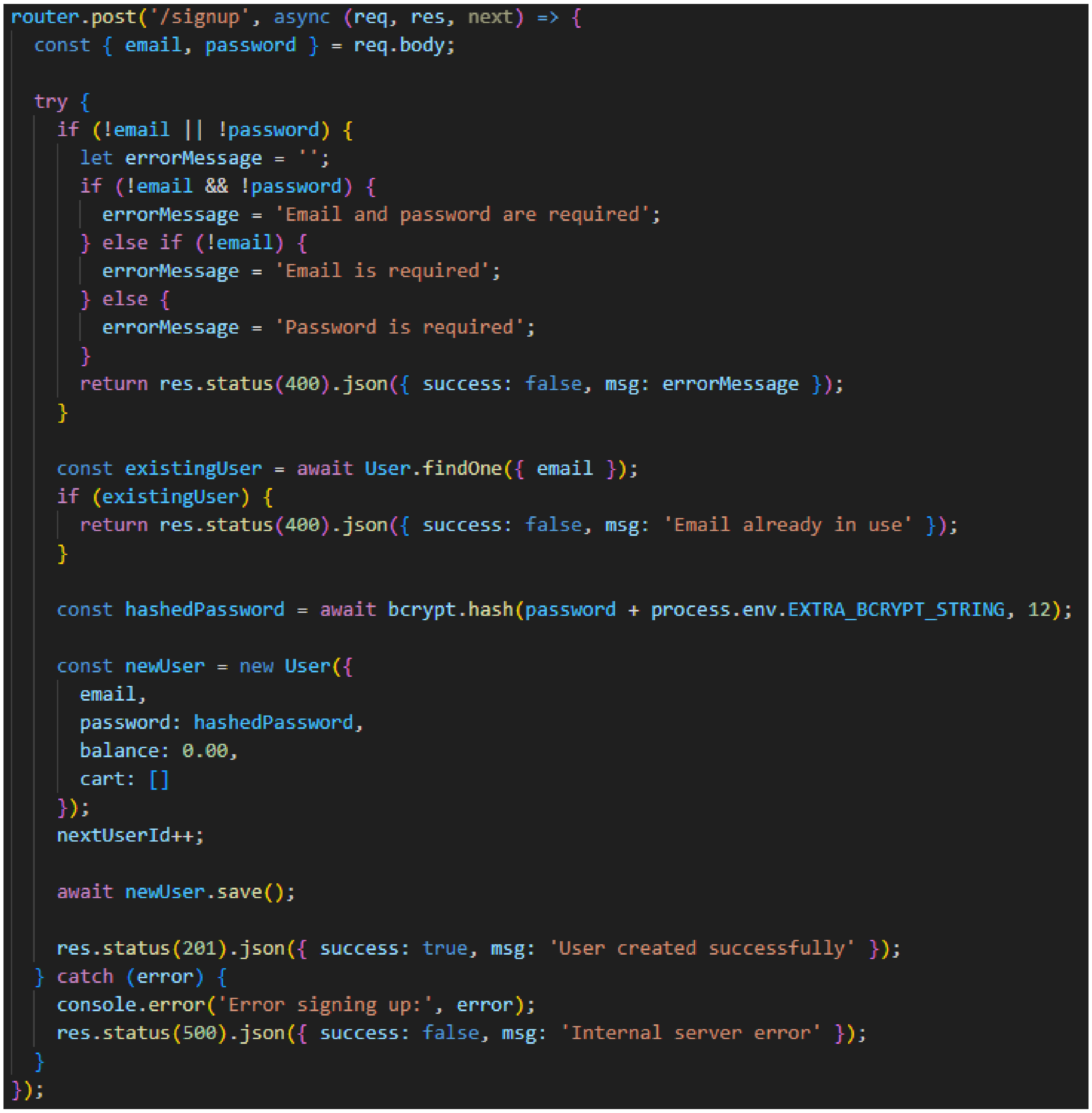


```
const itemSchema = new Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true },
  stock: { type: Number, required: true },
  productCode: { type: String, required: true }
});

const Item = mongoose.model('Item', itemSchema);
```

Figure 8-4 Item Schema

In my back end I have 2 schemas defined and they both serve a different purpose but work hand in hand. The user Schema is used for my Login and Register endpoints. On registration or login, the user is required to provide an email and password. Each user has a purchases array that stores their purchase history, a tag used for their payment and a balance to display the user's balance.

The Item Schema manages all the products. Each has a name, price, amount in stock and a product code. The product code is the Electronic Product Code (EPC) acquired from the RFID tags and this is how the products are identified in the database.

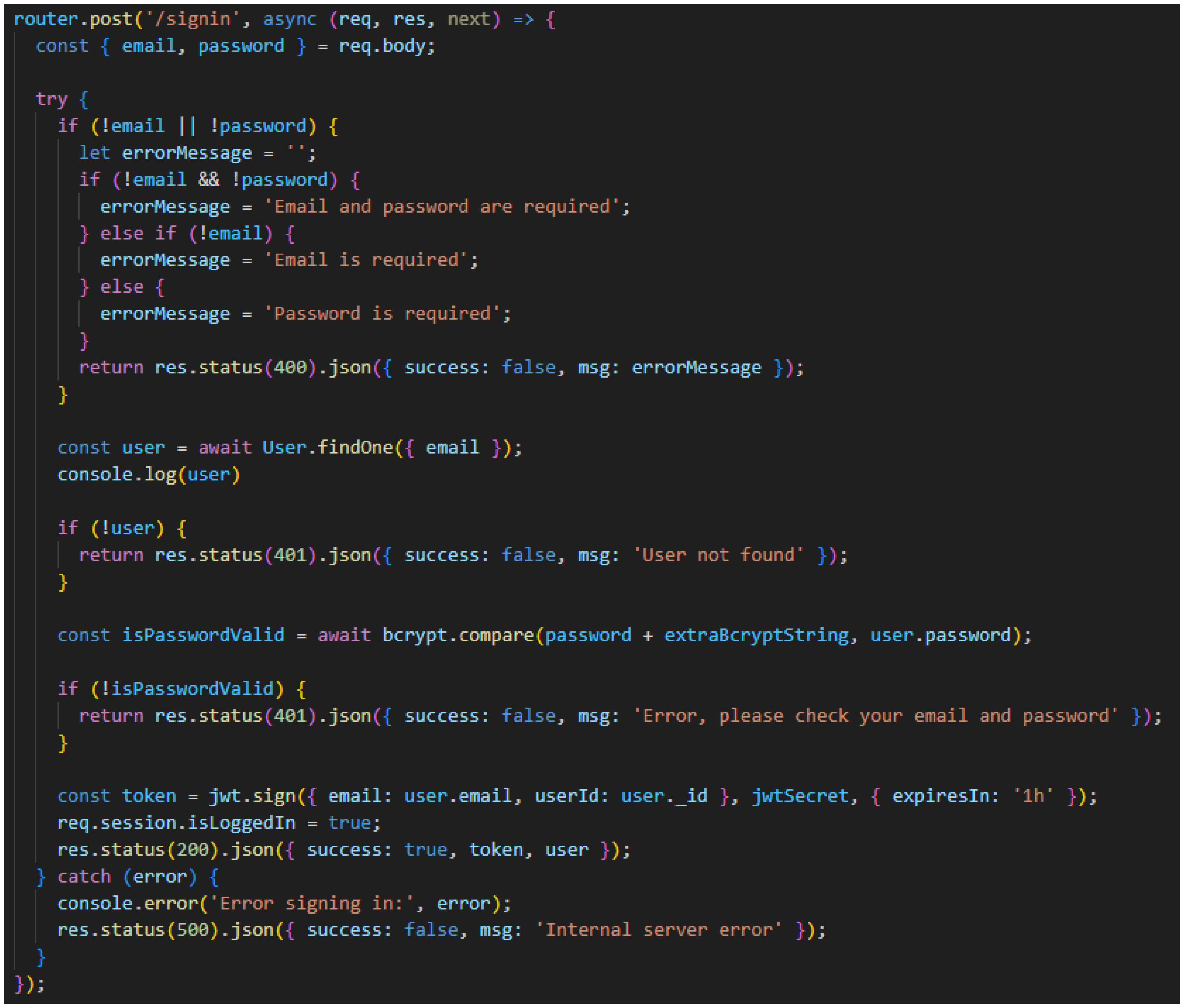


```
router.post('/signup', async (req, res, next) => {
  const { email, password } = req.body;

  try {
    if (!email || !password) {
      let errorMessage = '';
      if (!email && !password) {
        errorMessage = 'Email and password are required';
      } else if (!email) {
        errorMessage = 'Email is required';
      } else {
        errorMessage = 'Password is required';
      }
      return res.status(400).json({ success: false, msg: errorMessage });
    }

    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ success: false, msg: 'Email already in use' });
    }

    const hashedPassword = await bcrypt.hash(password + process.env.EXTRA_BCRYPT_STRING, 12);

    const newUser = new User({
      email,
      password: hashedPassword,
      balance: 0.00,
      cart: []
    });
    nextUserId++;

    await newUser.save();

    res.status(201).json({ success: true, msg: 'User created successfully' });
  } catch (error) {
    console.error('Error signing up:', error);
    res.status(500).json({ success: false, msg: 'Internal server error' });
  }
});
```

Figure 8-5 Sign up endpoint

When a POST request is made to the '**/signup**' endpoint, it expects the request body to contain an email and password. The code first checks if any fields have been left out and sends an appropriate message.

If the email is unique, it hashes the provided password using bcrypt, [12] which is a cryptographic hashing function. Bcrypt generates a secure hash of a password string and an additional string (called a "salt"). The salt is concatenated with the password before hashing. The number '12' is the cost factor, determining the computational cost of the hashing process.



```
router.post('/signin', async (req, res, next) => {
  const { email, password } = req.body;

  try {
    if (!email || !password) {
      let errorMessage = '';
      if (!email && !password) {
        errorMessage = 'Email and password are required';
      } else if (!email) {
        errorMessage = 'Email is required';
      } else {
        errorMessage = 'Password is required';
      }
      return res.status(400).json({ success: false, msg: errorMessage });
    }

    const user = await User.findOne({ email });
    console.log(user)

    if (!user) {
      return res.status(401).json({ success: false, msg: 'User not found' });
    }

    const isPasswordValid = await bcrypt.compare(password + extraBcryptString, user.password);

    if (!isPasswordValid) {
      return res.status(401).json({ success: false, msg: 'Error, please check your email and password' });
    }

    const token = jwt.sign({ email: user.email, userId: user._id }, jwtSecret, { expiresIn: '1h' });
    req.session.isLoggedIn = true;
    res.status(200).json({ success: true, token, user });
  } catch (error) {
    console.error('Error signing in:', error);
    res.status(500).json({ success: false, msg: 'Internal server error' });
  }
});
```

Figure 8-6 Sign in endpoint

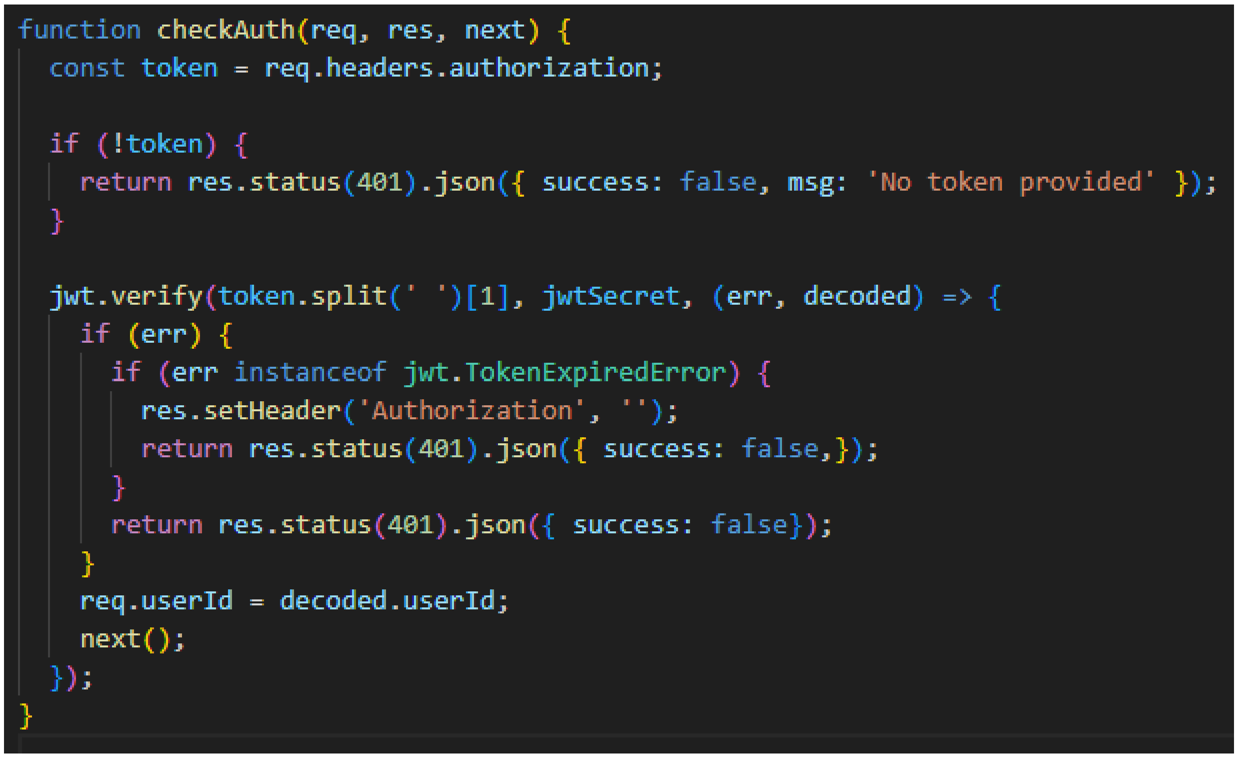
The **'/signin'** endpoint is similar to the sing up one. It expects an email and password and checks all fields.

Then the code queries the database to find a user with the provided email. If no user is found, a response is sent back.

If a user with the provided email exists, [13] it compares the hashed password stored in the database with the hashed version of the provided password. It uses bcrypt's compare function to securely compare the two hashed passwords. If the passwords match, the user is considered authenticated; otherwise, it returns a 401 status with an error message.

If the authentication is successful, a JSON Web Token (JWT) is generated using the jwt.sign function. A JWT is a compact, URL-safe means of representing claims to be transferred between two parties. In this case, the JWT contains the user's email and user ID, and it's signed with a secret key (jwtSecret) known only to the server. The ‘expiresIn’ option specifies the token's expiration time.

JWT allows the server to securely encode user information into a token, which can be sent to the client and included in subsequent requests to authenticate the user without the need to store user data on the server.



```
function checkAuth(req, res, next) {
  const token = req.headers.authorization;

  if (!token) {
    return res.status(401).json({ success: false, msg: 'No token provided' });
  }

  jwt.verify(token.split(' ')[1], jwtSecret, (err, decoded) => {
    if (err) {
      if (err instanceof jwt.TokenExpiredError) {
        res.setHeader('Authorization', '');
        return res.status(401).json({ success: false,});
      }
      return res.status(401).json({ success: false});
    }
    req.userId = decoded.userId;
    next();
  });
}
```

Figure 8-7 Check Authorization

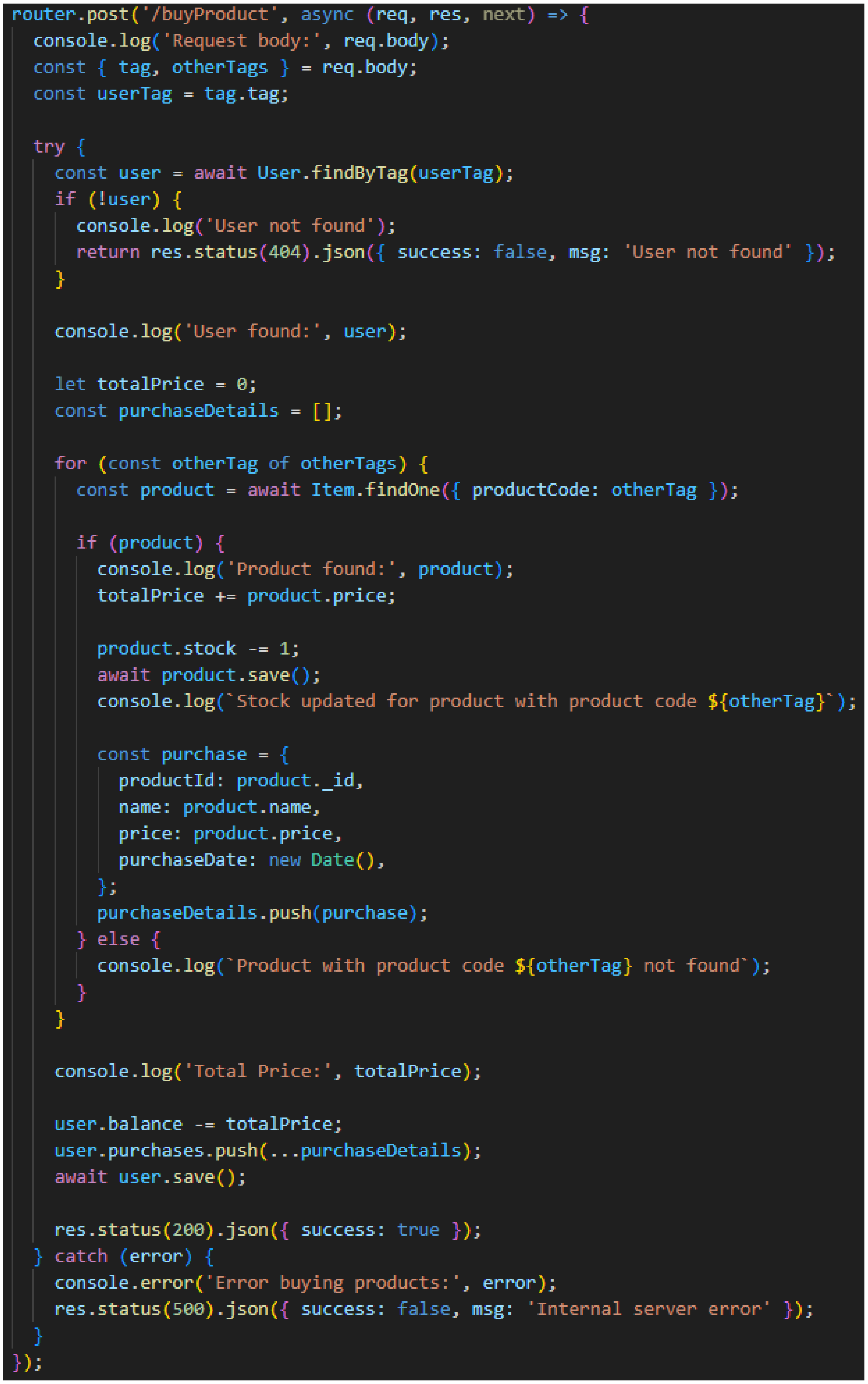
This is a middleware function used for authenticating requests by verifying JSON Web Tokens (JWTs).
The token is first extracted from the header and then checked if its empty. If token is provided it is verified using '**jwt.verify()**' function. If the token is invalid, a 401 error is returned, else the user ID is extracted from the decoded token payload and attached to the request object '**req.userId**'

[14]'**jwt.verify()**' has 3 parameters that need to be verified: The token, secret key, and a callback function. Decoding the token extracts the payload, which contains user information. Then the signature is checked to ensure it hasn't been tampered with.

If the token is valid and its signature matches, '**jwt.verify()**' invokes the callback function with **null** as the error and the decoded payload as the second argument.

If the token is invalid, such as being expired or malformed,' **jwt.verify()'** invokes the callback function with an error object describing the issue.

'**jwt.verify()**' operates asynchronously, meaning it doesn't block the execution of other code. Instead, it utilizes a callback function to handle the verification result once it's available.



```
router.post('/buyProduct', async (req, res, next) => {
  console.log('Request body:', req.body);
  const { tag, otherTags } = req.body;
  const userTag = tag.tag;

  try {
    const user = await User.findByTag(userTag);
    if (!user) {
      console.log('User not found');
      return res.status(404).json({ success: false, msg: 'User not found' });
    }

    console.log('User found:', user);

    let totalPrice = 0;
    const purchaseDetails = [];

    for (const otherTag of otherTags) {
      const product = await Item.findOne({ productCode: otherTag });

      if (product) {
        console.log('Product found:', product);
        totalPrice += product.price;

        product.stock -= 1;
        await product.save();
        console.log(`Stock updated for product with product code ${otherTag}`);

        const purchase = {
          productId: product._id,
          name: product.name,
          price: product.price,
          purchaseDate: new Date(),
        };
        purchaseDetails.push(purchase);
      } else {
        console.log(`Product with product code ${otherTag} not found`);
      }
    }

    console.log('Total Price:', totalPrice);

    user.balance -= totalPrice;
    user.purchases.push(...purchaseDetails);
    await user.save();

    res.status(200).json({ success: true });
  } catch (error) {
    console.error('Error buying products:', error);
    res.status(500).json({ success: false, msg: 'Internal server error' });
  }
});
```

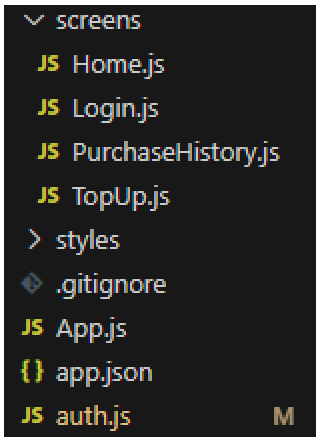
Figure 8-8 Buying Products endpoint

This is the endpoint that is called by the M5Atom Matrix. It is responsible for receiving the payment card information and products purchased by the user.

The ‘tag’ is the user payment card and is unique to the user. This tag is extracted from the request body and a function is called to find the user by this tag. ‘**otherTags’** are the products that are stored in the database. When a user is found, a ‘**for’** loop iterates through the products and looks for them in the database. Each product is updated in the database, a new total is calculated. Finally, the users balance is updated along with their purchase history.

8.3 Front end

The frontend of a software application is the part that users directly interact with. It's essentially the user interface—the visual elements, controls, and layouts that users see and interact with on their devices.



```
screens
  Home.js
  Login.js
  PurchaseHistory.js
  TopUp.js
styles
.gitignore
App.js
app.json
auth.js                M
```

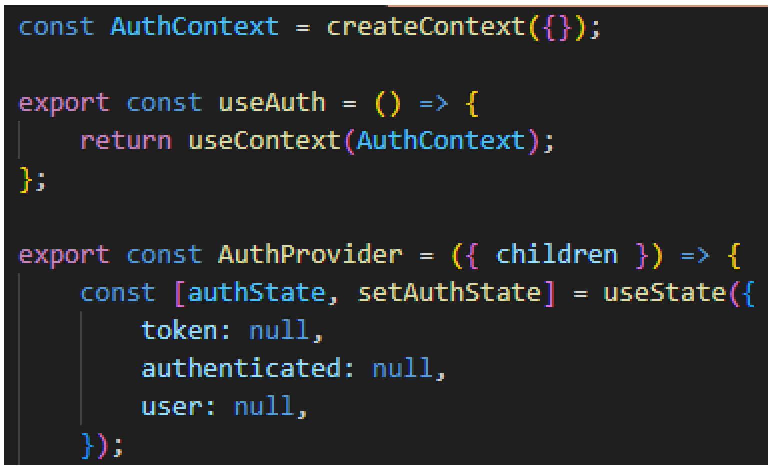
Figure 8-9 Application structure

The front end of my application is divided into separate folders and JavaScript files to make it easy to navigate around and modify code.

The main part of my front end is the ‘**auth.js**’ file. This file directly interacts with the server and provides a centralized mechanism for managing user authentication and authorization within my application, facilitating user login, registration, logout, and authentication state management.

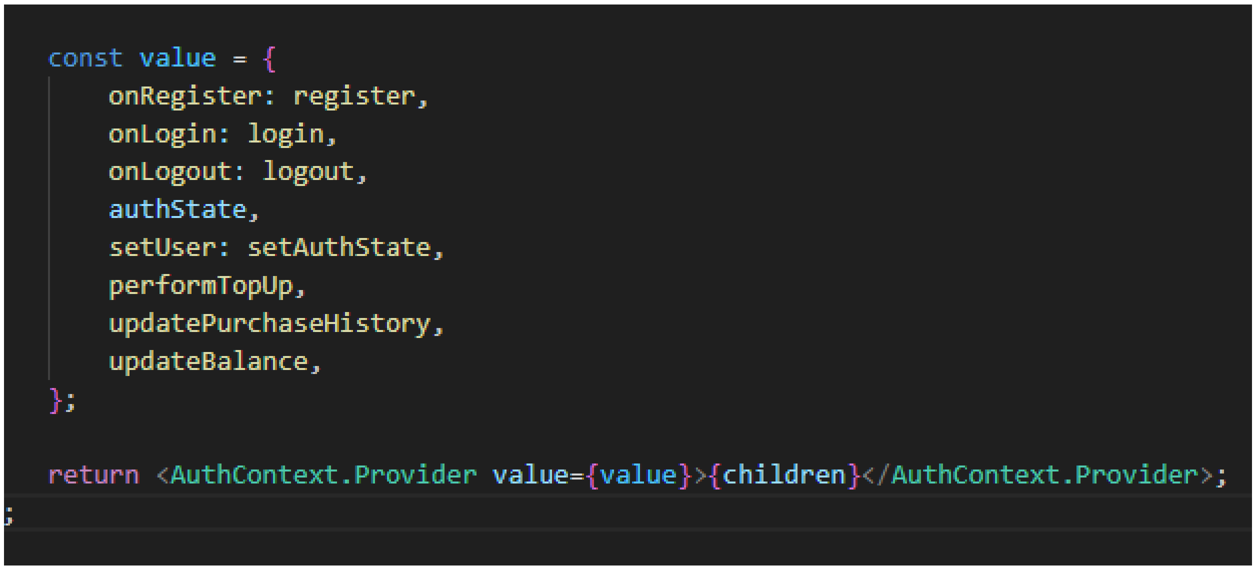
8.3.1 Auth.js

In this section I will break down the inner workings of the auth.js file



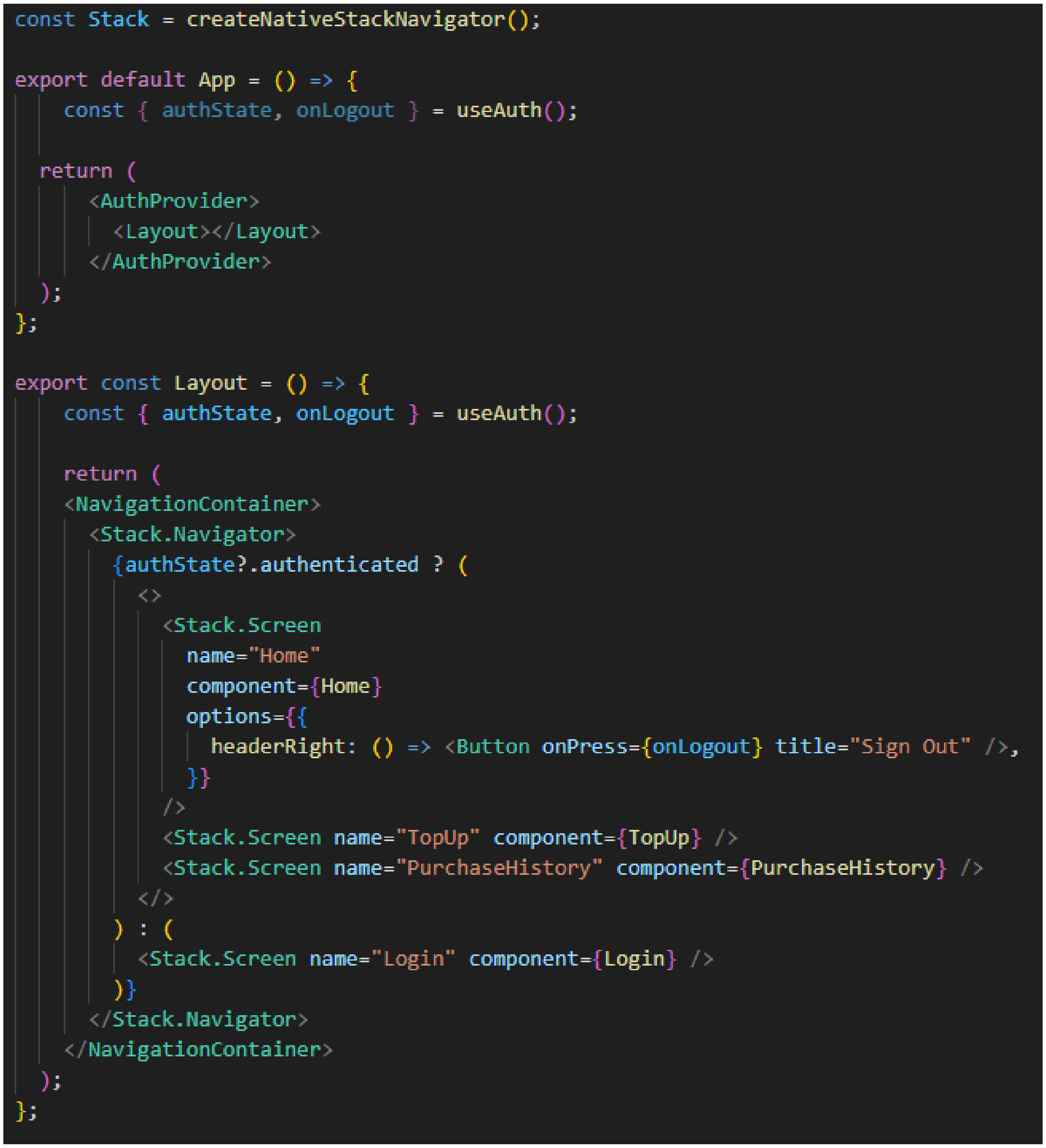
```
const AuthContext = createContext({});

export const useAuth = () => {
    return useContext(AuthContext);
};

export const AuthProvider = ({ children }) => {
    const [authState, setAuthState] = useState({
        token: null,
        authenticated: null,
        user: null,
    });
```

Figure 8-10 useContext / AuthProvider



```
const value = {
    onRegister: register,
    onLogin: login,
    onLogout: logout,
    authState,
    setUser: setAuthState,
    performTopUp,
    updatePurchaseHistory,
    updateBalance,
};

return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
;
```

Figure 8-11 AuthContext.Provider



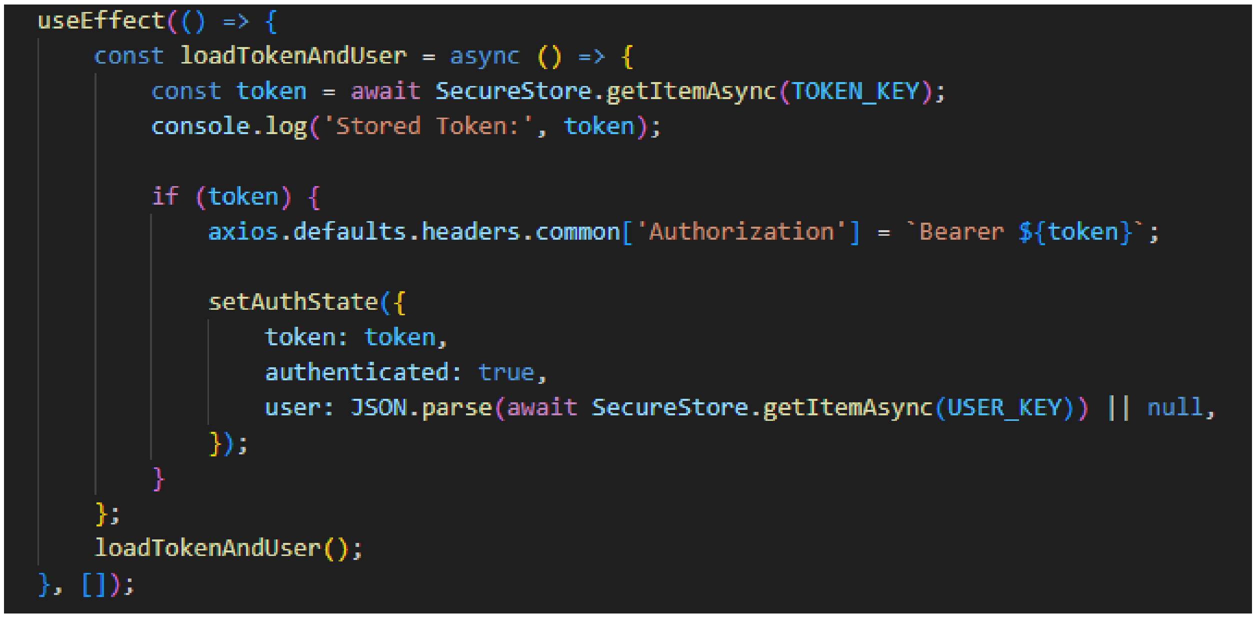
```
const Stack = createNativeStackNavigator();

export default App = () => {
    const { authState, onLogout } = useAuth();

  return (
    <AuthProvider>
      <Layout></Layout>
    </AuthProvider>
  );
};

export const Layout = () => {
    const { authState, onLogout } = useAuth();

    return (
    <NavigationContainer>
      <Stack.Navigator>
        {authState?.authenticated ? (
          <>
            <Stack.Screen
              name="Home"
              component={Home}
              options={{
                headerRight: () => <Button onPress={onLogout} title="Sign Out" />,
              }}
            />
            <Stack.Screen name="TopUp" component={TopUp} />
            <Stack.Screen name="PurchaseHistory" component={PurchaseHistory} />
          </>
        ) : (
          <Stack.Screen name="Login" component={Login} />
        )}
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Figure 8-12 AuthProvider in App.js

The ‘**useAuth’** hook is a custom React hook used for accessing authentication-related data and functionalities within functional components of my application. It provides a convenient way to access the authentication state and perform authentication-related actions without having to pass props or use context directly.

The ‘**AuthContext**’ is created using ‘**createContext()**’. This context serves as a central place to store and share authentication-related data and functions across components.

The '**AuthProvider**' component wraps the entire application as seein in Figure 8-11. It initializes and manages the authentication state using ‘**useState()**’ and **‘useEffect()**’ hooks. It also exposes authentication-related functionalities through the context value provided by ‘**AuthContext.Provider**’ as seen in Figure 8-12.



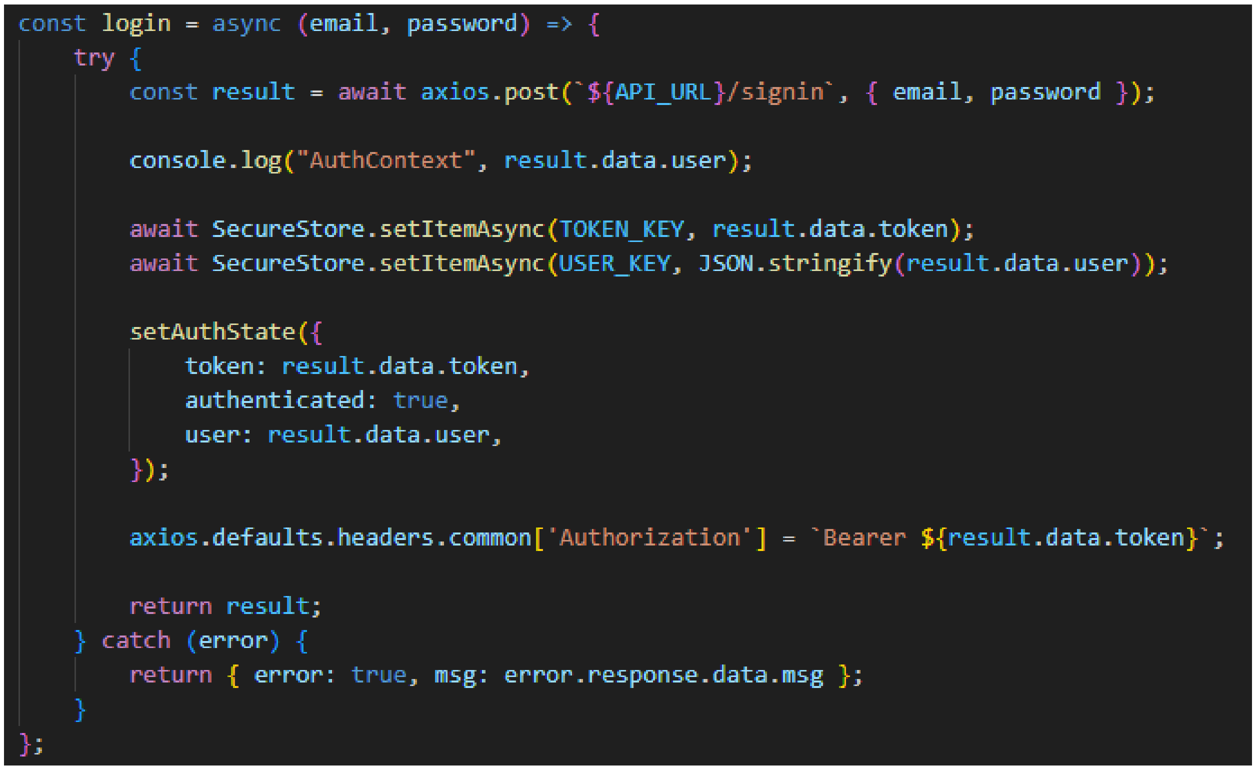
```
useEffect(() => {
    const loadTokenAndUser = async () => {
        const token = await SecureStore.getItemAsync(TOKEN_KEY);
        console.log('Stored Token:', token);

        if (token) {
            axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;

            setAuthState({
                token: token,
                authenticated: true,
                user: JSON.parse(await SecureStore.getItemAsync(USER_KEY)) || null,
            });
        }
    };
    loadTokenAndUser();
}, []);
```

Figure 8-13 loadTokenAndUser

The ‘**loadTokenAndUser**’ function is an asynchronous function responsible for loading the authentication token and user data from the device's secure storage when the ‘**AuthProvider**’ component mounts.

The ‘**loadTokenAndUser’** function is invoked inside the ‘**useEffect’** hook of the ‘**AuthProvider’** component. This ensures that it runs when the component mounts, allowing the authentication token and user data to be loaded asynchronously and updated in the component state ‘**authState**’.

By separating the token and user data loading logic into a dedicated function and using asynchronous operations, the code promotes readability, maintainability, and robust error handling. Additionally, it ensures that the authentication state is properly initialized when the component mounts, providing a seamless user experience.

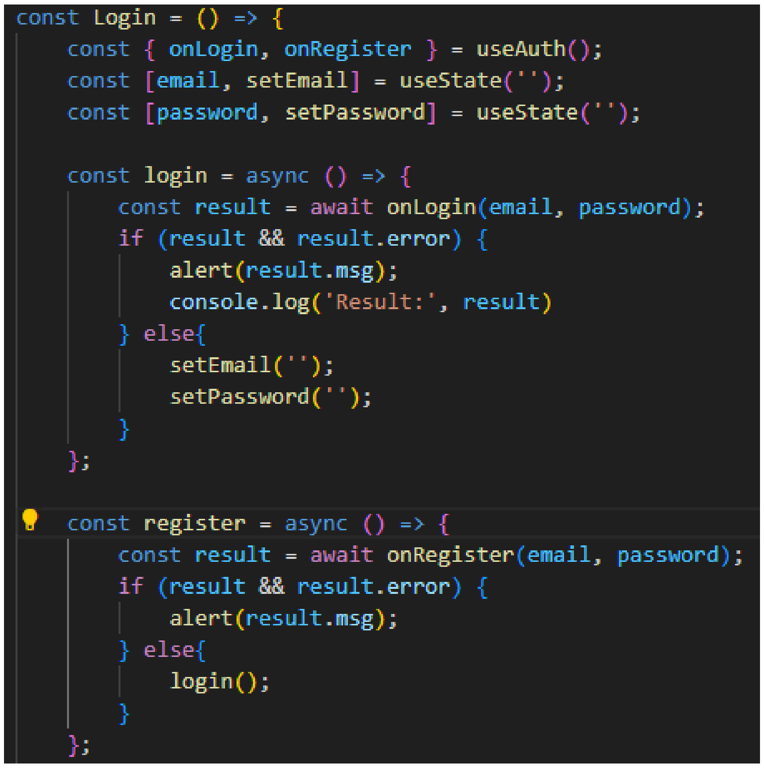


```
const login = async (email, password) => {
    try {
        const result = await axios.post(`${API_URL}/signin`, { email, password });

        console.log("AuthContext", result.data.user);

        await SecureStore.setItemAsync(TOKEN_KEY, result.data.token);
        await SecureStore.setItemAsync(USER_KEY, JSON.stringify(result.data.user));

        setAuthState({
            token: result.data.token,
            authenticated: true,
            user: result.data.user,
        });

        axios.defaults.headers.common['Authorization'] = `Bearer ${result.data.token}`;

        return result;
    } catch (error) {
        return { error: true, msg: error.response.data.msg };
    }
};
```

Figure 8-14 login function

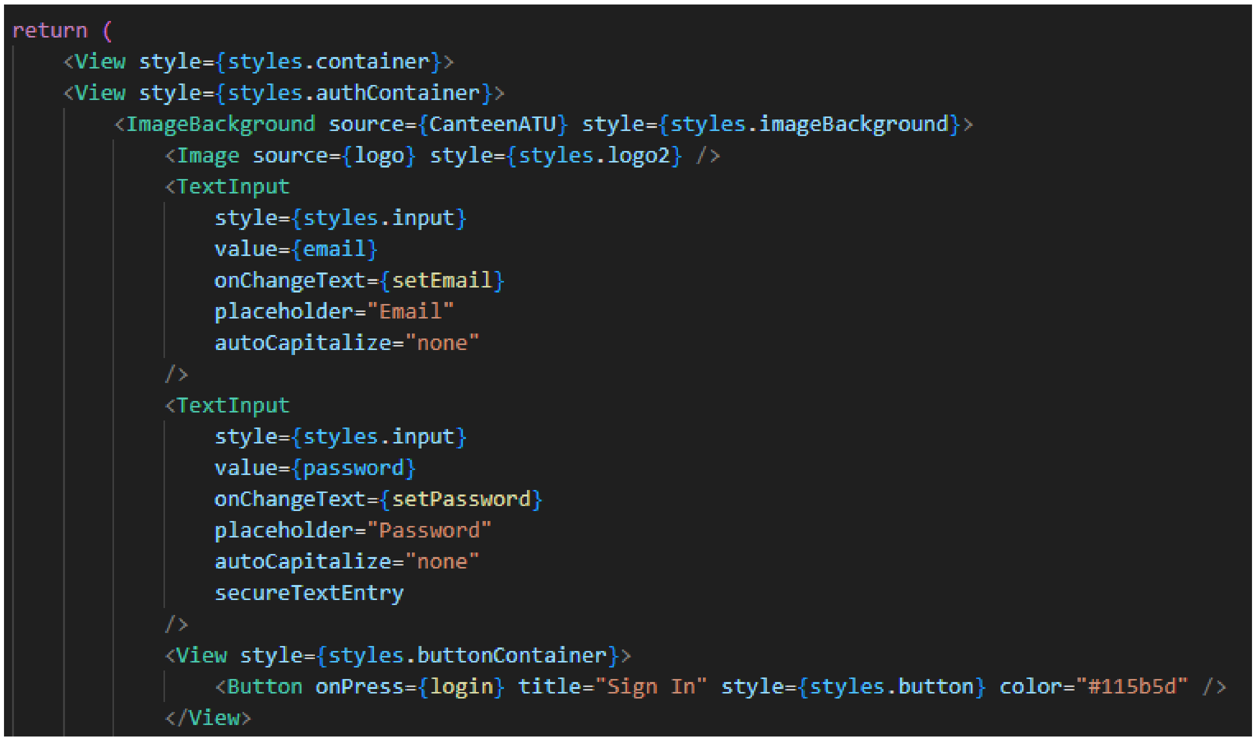
In the auth.js file I have several other functions like register, logout, performTopUp, updatePurchaseHistory, and updateBalance. All the functions work in a similar fashion explained below.

My ‘**login**’ function is responsible for authenticating users by sending a login request to the server with provided credentials. Upon successful authentication, the token and user data are securely stored on the device using '**SecureStore.setItemsAsync**’ method. The authentication state is updated with the obtained token and user data. The '**axios.default.header.common**' sets the default authorization header for Axios requests to include the obtained token, ensuring subsequent requests are authenticated



```
const Login = () => {
    const { onLogin, onRegister } = useAuth();
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');

    const login = async () => {
        const result = await onLogin(email, password);
        if (result && result.error) {
            alert(result.msg);
            console.log('Result:', result)
        } else{
            setEmail('');
            setPassword('');
        }
    };

    const register = async () => {
        const result = await onRegister(email, password);
        if (result && result.error) {
            alert(result.msg);
        } else{
            login();
        }
    };
```

Figure 8-15 Login and register functions



```
return (
    <View style={styles.container}>
    <View style={styles.authContainer}>
        <ImageBackground source={CanteenATU} style={styles.imageBackground}>
            <Image source={logo} style={styles.logo2} />
            <TextInput
                style={styles.input}
                value={email}
                onChangeText={setEmail}
                placeholder="Email"
                autoCapitalize="none"
            />
            <TextInput
                style={styles.input}
                value={password}
                onChangeText={setPassword}
                placeholder="Password"
                autoCapitalize="none"
                secureTextEntry
            />
            <View style={styles.buttonContainer}>
                <Button onPress={login} title="Sign In" style={styles.button} color="#115b5d" />
            </View>
```

Figure 8-16 Login form

The ‘**Login**’ component uses the ‘**useState**’ hook to manage state variables for ‘**email’** and ‘password’, representing the user’s input.

There are 2 functions in here: ‘**login’** and ‘**register**’ which are responsible for handling the login and registration processes, respectively. These functions interact with the '**onLogin'** and '**onRegister'** functions from '**auth.js'** to perform authentication-related tasks.

When the ‘**Sign in**’ button is pressed it triggers the ‘**login**’ function. This function calls ‘**onLogin**’ from ‘**auth.js**’ with the provided email and password. If authentication fails, an alert message is displayed. Otherwise, the email and password inputs are cleared, and the user is brought to the ‘**Home**’ screen where they can perform operations such as account top up and view their past purchases

9 Ethics

Secure Data Transmission: In my project I had to consider sensitive information such as email and password. To keep the data secure when it's being sent to the server, I use POST requests. POST requests are more secure than GET requests as they encrypt sensitive data is sent in the request body rather than as part of the URL.

Storage Encryption: User passwords are one of the most sensitive pieces of information in my project, those are securely stored using encryption techniques. Passwords are hashed using bcrypt before being stored in the database. Bcrypt is a secure hashing algorithm designed to protect passwords from being compromised in the event of a data breach. This ensures that even if the database is compromised, passwords cannot be easily decrypted or reverse engineered.

Secure Authentication: When users log in to the application, their credentials are verified securely using bcrypt. Bcrypt compares the hashed password stored in the database with the hashed version of the password provided by the user during login. This process ensures that passwords are never stored or transmitted in plain text, reducing the risk of unauthorized access or password theft.

Secure Storage on Client Side: User authentication tokens and user data are securely stored on the client-side using SecureStore. SecureStore is a local storage mechanism provided by Expo that encrypts data before storing it on the device. This ensures that sensitive information such as authentication tokens and user data are protected from unauthorized access even if the device is lost or stolen.

10 Conclusion

Overall, I'm happy with how the project has turned out. I managed to implement the most important functionalities of the project. In the beginning I used MySQL as my database and SpringBoot to communicate with it, but over time my project evolved and I decided that the best course of action is to keep everything on one Node.js server as it was much more efficient that way, rather than having my SpringBoot talking to the database then to the Node.js server and back to my App. There was just too much moving parts and too much room for error. I'm glad I have chosen Node.js as my server.

The App turned out as I expected with security features implemented to keep the user data and password safe. I wasn't expecting for my server to be running on AWS but I'm glad we learned about it in our Cloud Computing module, and I was able to integrate it into my project.

The M5Stack was a good choice for its modular architecture, allowing me to easily connect their RFID Reader and interact with it. Although, I've had difficulties with writing data to the tags, I've manged to work around it and still complete my project.

11 References

[1] B. Lin-Fisher, "eu.usatoday.com," USA TODAY, 04 04 2024. [Online]. Available: https://eu.usatoday.com/story/money/shopping/2024/04/04/amazon-just-walk-out-indian-workers/73204975007/. [Accessed 27 04 2024].

[2] Arduino Team, "arduino.cc," Arduino, 2024. [Online]. Available: https://www.arduino.cc/en/software. [Accessed 27 04 2024].

[3] M5Stack, "m5stack.com," M5Stack, 2024. [Online]. Available: https://shop.m5stack.com/products/atom-matrix-esp32-development-kit#:~:text=ATOM%20Matrix%20is%20a%20compact,ample%20flexibility%20for%20your%20projects.. [Accessed 27 04 2024].

[4] M5Stack, "m5stack.com," M5Stack, 2024. [Online]. Available: http://docs.m5stack.com/en/unit/uhf_rfid. [Accessed 27 04 2024].

[5] Wikipedia, "wikipedia.org," Wikipedia, 22 04 2024. [Online]. Available: https://en.wikipedia.org/wiki/React_Native. [Accessed 27 04 2024].

[6] S. S. P. Limited, "medium.com," 27 06 2023. [Online]. Available: https://medium.com/@softworthsolutionspvtltd/expo-vs-react-native-cli-7e47c7630039. [Accessed 27 04 2024].

[7] Wikipedia, "wikipedia.org," 12 04 2024. [Online]. Available: https://en.wikipedia.org/wiki/Express.js. [Accessed 28 04 2024].

[8] Wikipedia, "wikipedia.org," 26 04 2024. [Online]. Available: https://en.wikipedia.org/wiki/Node.js. [Accessed 28 04 2024].

[9] MongoDB, "mongodb.com," 28 04 2024. [Online]. Available: https://www.mongodb.com/. [Accessed 28 04 2024].

[10] N. Karnik, "freecodecamp.org," 11 02 2018. [Online]. Available: https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/. [Accessed 28 04 2024].

[11] Amazon, "aws.amazon.com," 2024. [Online]. Available: https://aws.amazon.com/ec2/. [Accessed 28 04 2024].

[12] Wikipedia, "wikipedia.org," 20 04 2024. [Online]. Available: https://en.wikipedia.org/wiki/Bcrypt#:~:text=The%20bcrypt%20function%20is%20the,distributions%20such%20as%20SUSE%20Linux.. [Accessed 28 04 2024].

[13] D. Arias, "auth0.com," 25 02 2021. [Online]. Available: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/. [Accessed 28 04 2024].

[14] Auth0, "npmjs.com," 08 2023. [Online]. Available: https://www.npmjs.com/package/jsonwebtoken#jwtverifytoken-secretorpublickey-options-callback. [Accessed 29 04 2024].

[15] M5Stack, "m5stack.com," M5Stack, 2024. [Online]. Available: https://m5stack.com/about-us. [Accessed 25 04 2024].