VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Pham Huy Quang

ONLINE BANKING WEB APPLICATION

A Study Case of Implementing a Banking Web Application

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Pham Huy Quang
Title	Online Banking Web Application
Year	2022
Language	English
Pages	50 + 4 References and Appendices
Name of Supervisor	Timo Kankaanpää

The main objective of this thesis is to create an elementary implementation of an online banking application, which are heavily used by present-day banking organizations world-wide. By studying examples from real world running instances of e-banking application, this thesis tried to achieve the characteristics and functionalities of a typical web based online banking software.

Using popular frameworks and technologies in the web development field such as Typescript, React, Express, PostgreSQL and many more the project will combine common features of a web applications, such as login-logout registration, user-based data management and interactions with basic financial solutions that the normal user can expect from a real-world banking website such as view simulated financial data about account, conducting transactions and view bank statements.

The application will be deployed onto the cloud and can be tested by anyone that has access to the website address, approximately under one-year time frame. This will be the first version of the application, acting as a practical argument for topic of this paper.

Keywords Web development, web application

CONTENTS

FIGURES, DIAGRAMS AND TABLES LIST

1 INTRODUCTION

1.1 Background & Motivations

The exchange of goods, products and services has coexisted with mankind since about 1800 BC in ancient Babylonia. From a humble beginning such as using a simple table as a makeshift desk where goods bartering and local commerce were conducted; which formed the word "bank"; financial and monetary transactions has become a common activity that take place probably every second. Furthermore, transactional communications have evolved with an exponential speed that it now has accepted: from valuable items such as silver and gold ingots, to metal and paper currency, to now digitally verifiable records that are stored on the clouds as payments for goods and services. With the help of a computers and the speed of information exchange brought by technologies, monetary authority and banking institutions can even utilize these advantages to let customers and citizens conduct a variety of financial transactions "just with a click of a button".

Personal computers and smartphones have revolutionized the idea of personal banking and brought safety, convenience and speed to the matter. For instance, a relative could send money to his loved one through an e-banking application without the need to go to his local bank's branch and notify the employee with his documents; which may take probably one to two days at maximum. With the current Corona pandemic, the world is facing right now that limited mobility, the benefits of using digital technologies for monetary transactions are tremendous and undeniable. Having studied in the web development field and have been familiarized with banking concepts from a family member, the author took interested in the matter he mentioned above. A challenging opportunity rose to showcase his knowledge and proficiency in web related programming languages, libraries and framework by implementing a web-based application with e-banking characteristics. By using ideas and inspirations from real world examples of running e-banking web programs, a web-based banking software was built.

1.2 Objectives

The objective is to create a web application with a typical online banking characteristic. Table 1 below lists all the necessary features of the application.

Table 1. Main objectives of the project.

Use Case	Basic Flow
Register	User fill in the form with validated data to create a User Account.
Login	User fill in the form with validated data to be able to access protected resources of the application.
Logout	User clicks the Logout button to destroy the session and revoking his rights to access protected resources.
Reload Session	By defaults, User can only use the site for a short time (e.g., 10 minutes). User can click the Reload button to confirm the User is actively using the website and won’t be forced logout.
Authentication and Authorization	User must login if he wishes to access protected resources of the application. This is done using session ID and/or user ID.
View account data	User can view and edit his User Information.

Earn money	User can click Simulate Income Earning to have his bank account's balance increase as a mock-up way to represent earning credits in real-life bank accounts.
Send a transaction	User can perform a financial transaction by fill in a form with validated data.
Receive a transaction	User can see his balance changed and new transactions if he is involved in a financial transaction.
View bank statements	User can list all the Financial Transaction records that related to him.

Some common grounds are needed to be established in this session to further avoid misinterpretation. When using the application, the user:

- will interact with the front-end side of the application through a web browser, Firefox latest edition recommended, on a Windows machine
- will use mouse the main action initiator when performing operations through the front-end side
- All actions performed in this application are handled in the HTTP application layer with its core methods are GET, POST, PUT and DELETE

1.3 Scope & Limitations

Unfortunately, considering the scope of this thesis's project, not all real life features could be included in the thesis. Below are the limitations regarding the contents of the applications:

- some common HTTP actions will not be implemented on some features of the application due to the limits of time, the complexity or the inexperienced of the author
- using an account as a way for identifications from third parties
- performing any kind of external transactions to other accounts that is not created in this project (e.g., transferring to other real-life agencies, institutes and organizations)
- using this program as a mobile web application

1.4 Thesis Structure

Following a common pattern when building software, the thesis's work is divided into two separable parts that communicate with each other through the HTTP protocol to form a whole web application experience.

Frontend (FE for short) acts as a presentation layer of the application. The FE of the application provides graphical user interface of the website for user interactions and - manages communications between itself and the BE to form the flow of data.

Backend (BE for short) acts as a data access layer while also handling requests and code flow logic of the application. The BE of the application - receives HTTP requests from the FE, processes those requests, applies changes, runs logic code and returns respective responses to FE and interacts and mutates data that would be read, modified and written in databases and finally, provides basic securities measures to partially imitate real world solutions.

2 REQUIREMENTS & PLANNING

2.1 Study Cases of Real-world Examples

Two study cases of e-banking applications are presented below that are currently run and maintained by real-life banking organizations: Danske Bank A/S and Eximbank of Vietnam. All the images, medias, trademarks and intellectual properties used in this section rightfully belong to their respective owners with all rights reserved.

In the case of e-bank application from Danske Bank, only some of the most common pages are included to showcase the core points of functionalities which a regular customer would use an online banking application. These pages include a public Login Page, three private pages that are the Home Page, Statement Page and the Transaction Page.

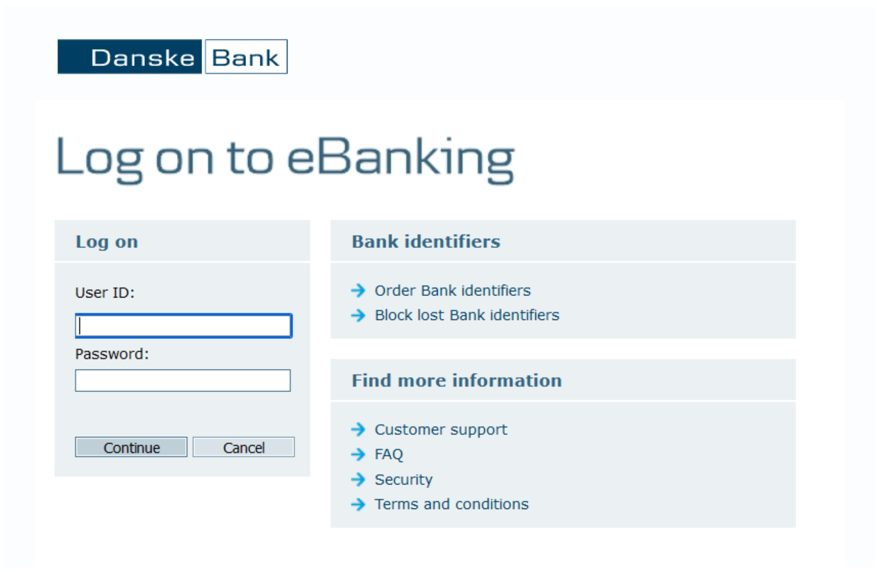


Figure 1. Danske Login Page with a small form.

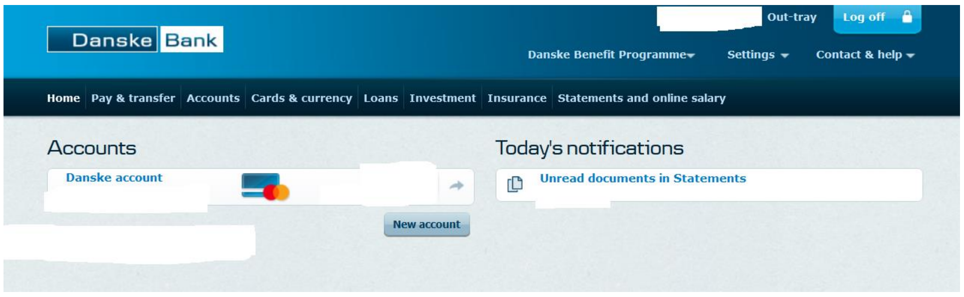
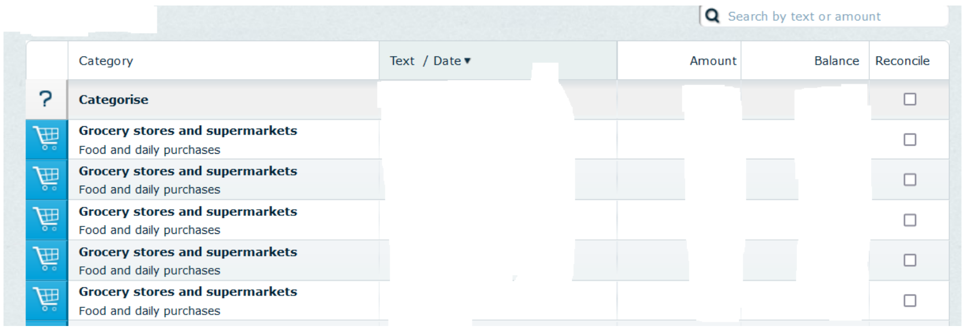
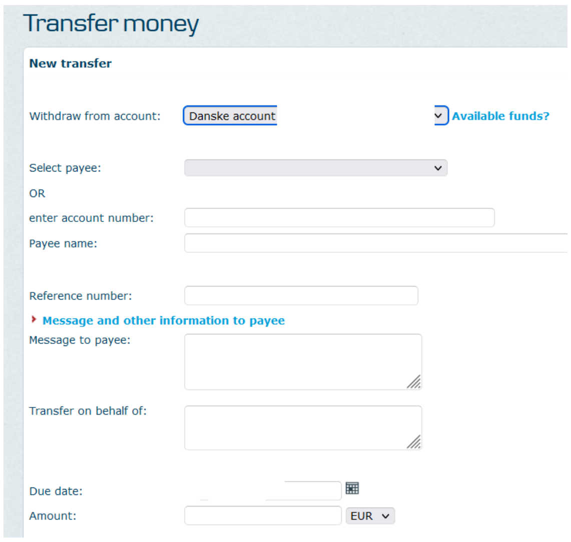


Figure 2. Danske User Home Page.


Search by text or amount

	Category	Text / Date ▾	Amount	Balance	Reconcile
?	**Categorise**	[redacted]	[redacted]	[redacted]	☐
🛒	**Grocery stores and supermarkets** Food and daily purchases	[redacted]	[redacted]	[redacted]	☐
🛒	**Grocery stores and supermarkets** Food and daily purchases	[redacted]	[redacted]	[redacted]	☐
🛒	**Grocery stores and supermarkets** Food and daily purchases	[redacted]	[redacted]	[redacted]	☐
🛒	**Grocery stores and supermarkets** Food and daily purchases	[redacted]	[redacted]	[redacted]	☐
🛒	**Grocery stores and supermarkets** Food and daily purchases	[redacted]	[redacted]	[redacted]	☐

Figure 3. Danske User Statement Page.

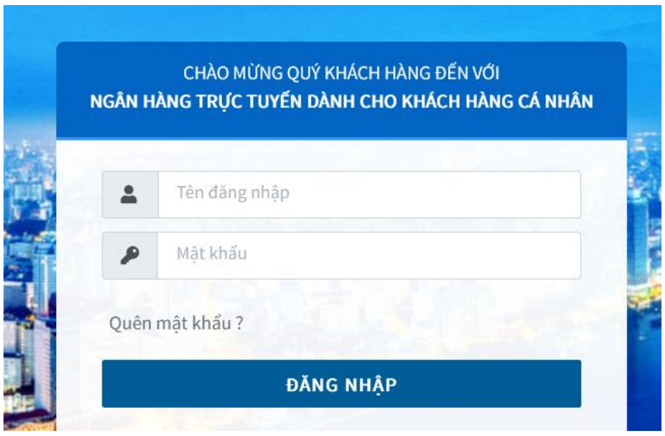

Transfer money

New transfer

Withdraw from account:	Danske account
Select payee:	
OR	
enter account number:	
Payee name:	
Reference number:	
Message to payee:	
Transfer on behalf of:	
Due date:	
Amount:	EUR

Figure 4. Danske User Transaction Page.

The case of e-bank application from Eximbank gives another example of how a proprietary software should be implemented and developed. While each example is taken from different countries, and fall under different jurisdictions and regulations, similarities in designing and behaviors can be seen in both applications. From Figure 5 to Figure 8, again, shows the common pages that users would use in an online banking application, which are the Login Page, the Home Page, the Statement Page and the Transaction Page.


CHÀO MỪNG QUÝ KHÁCH HÀNG ĐẾN VỚI
NGÂN HÀNG TRỰC TUYẾN DÀNH CHO KHÁCH HÀNG CÁ NHÂN

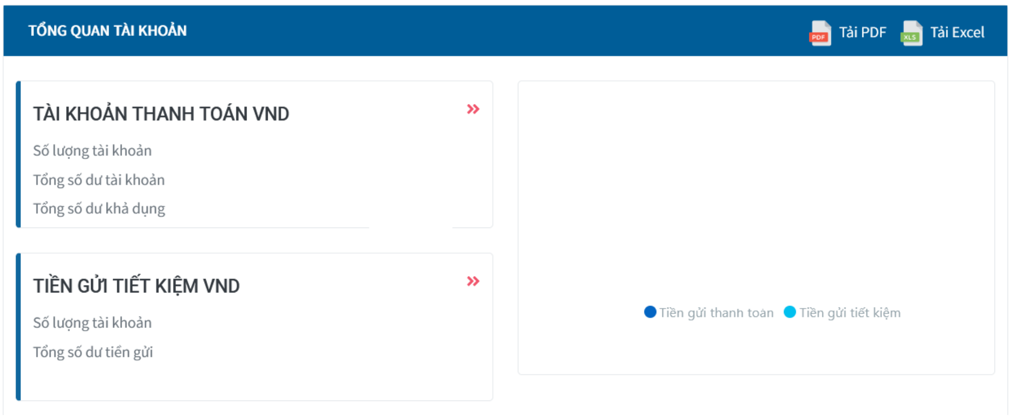
Tên đăng nhập

Mật khẩu

Quên mật khẩu ?

ĐĂNG NHẬP

Figure 5. Eximbank Login Page with two input fields.


TỔNG QUAN TÀI KHOẢN — Tải PDF — Tải Excel

TÀI KHOẢN THANH TOÁN VND »
Số lượng tài khoản
Tổng số dư tài khoản
Tổng số dư khả dụng

TIỀN GỬI TIẾT KIỆM VND »
Số lượng tài khoản
Tổng số dư tiền gửi

● Tiền gửi thanh toan ● Tiền gửi tiết kiệm

Figure 6. Eximbank User Home Page show only the most crucial data.

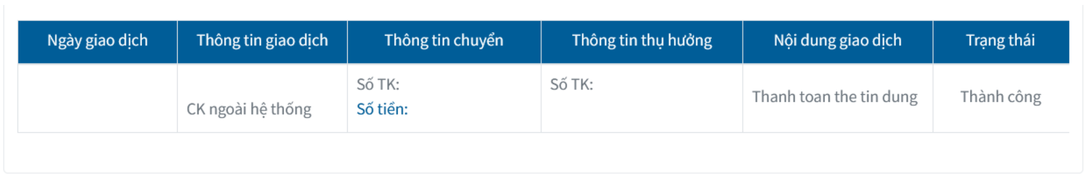
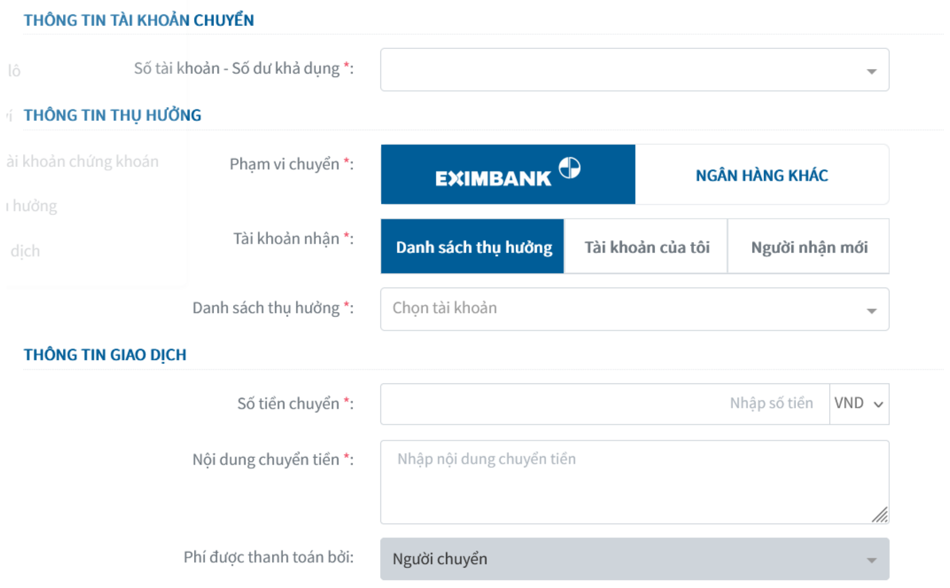

Ngày giao dịch	Thông tin giao dịch	Thông tin chuyển	Thông tin thụ hưởng	Nội dung giao dịch	Trạng thái
	CK ngoài hệ thống	Số TK: Số tiền:	Số TK:	Thanh toan the tin dung	Thành công

Figure 7. Eximbank User Statement Page.



THÔNG TIN TÀI KHOẢN CHUYỂN

Số tài khoản - Số dư khả dụng *:	

THÔNG TIN THỤ HƯỞNG

Phạm vi chuyển *:	EXIMBANK / NGÂN HÀNG KHÁC
Tài khoản nhận *:	Danh sách thụ hưởng / Tài khoản của tôi / Người nhận mới
Danh sách thụ hưởng *:	Chọn tài khoản

THÔNG TIN GIAO DỊCH

Số tiền chuyển *:	Nhập số tiền VND
Nội dung chuyển tiền *:	Nhập nội dung chuyển tiền
Phí được thanh toán bởi:	Người chuyển

Figure 8. Eximbank User Transaction Page.

Having observed and studied how a real-world online banking application are researched and developed, some key points and concepts were found that should be implemented in this application since these features are the minimum requirements of how a banking web application should be defined and behave.

In both examples, the users should be able to view the general information about themselves and their accounts. The users should also be able to order a bank statement that lists down every financial transaction that they are related to (received money or transfer money), as well as be able to submit data to the server to create a transaction. All the sensitive information should be protected and can only be viewed, and interacted if the user’s credentials are proven and authorized.

2.2 Technical Background

This segment is dedicated to briefly explain the concepts and the usability of programming technologies that the author would use to implemented in his application.

(TypeScript, 2022) is a modern programming language that contributed a large part in building the World Wide Web. It is structured based on the ECMAScript standard, this high-level language provides dynamic typings, prototype-based object-orientation and first-class functions. It is considered a popular choice when building web-related products like websites, online application and so on. Microsoft, seeing the potential growth, developed Typescript as a strict syntactical superset of JavaScript, improving the possibilities Javascript could bring on the development world, while trying to fix some of Javascript drawbacks like its ease of errors, unsafe typings and clearer programming instructions using its rich system of types and interface. One which looking for works in the field of web development is recommended to learn Typescript.

(React, 2022) is a free and open-source front-end Javascript library for building user-interface with their principles of component-based elements. By dividing a website into many small entities called "component", React can use the strategy "divide-and-conquer" to separate the points of concerns of each problem to each corresponding component. Each component should be written to be reusable and be rendered based on the code logic and handle all the necessary issues that are related to itself (e.g., fetching data from the third-party source, rendering based on properties being passed on)

(React Router, 2022) is considered as the second part of "bread-and-butter" while building web application with React. As the nature of React, which is being a single page application, React Router is created to solve the problems of routing in modern web app without ever refreshing the page when navigating.

(Context API, 2022) is a built-in state management system in React, making it integral when building websites with React, as it provides a way to manage complex and nested throughout the whole application.

(Styled-components, 2022) is a third-party open-source library giving the developers a way to create simple and reusable React components that can dynamically renders and adapting necessary behaviors based on passed properties to the components.

(NodeJS, 2022) is an open-source backend language written in Javascript using the V8 Chrome engine that can execute Javascript even outside of the familiar web browser environment. Born based on a humble paradigm of "making Javascript writable everywhere", NodeJS has played a huge part in unifying a web-application development around a single programming language, reducing the needs of learning another new language if one wish to choose NodeJS as the backend language.

(Express, 2022) is a backend web application framework built on the foundation of NodeJS. Under the MIT license, it is one of the popular choices in the industry when building server-side web applications and APIs, among other famous candidates such as NestJS and MeteorJS.

(Nginx, 2022) is, as quoted from the official documentation: an HTTP and reverse proxy server, originally written by Igor Sysoey. Standing firm at sharing 21.79% of hosting and running some of the busiest websites in the Internet, one could not argue of how much of Nginx have bought onto the table regarding the standardization of a HTTP server.

(PostgreSQL, 2022) is a free and open-source relational database management system, having one of its biggest selling points are the ACID properties: atomicity, consistency, isolation and durability. It is available in many operating systems such as Windows and Linux, and can perform a variety of workloads from single machines to a data warehouse or web services with many concurrent users.

(TypeORM, 2022) is a library built based on the data mapper pattern, which performs bidirectional transfer of data between a typical relational database and an in-memory data representation.

(AWS, 2022), short for Amazon Web Services, is an integration of many cloud services provided by Amazon, acting as a building block to create and deploy any types of applications or services in the cloud. Starting from a humble data storage cloud provider, AWS at the time of writing, provides on-demand delivery of technologies services and can almost power any customer's infrastructure regardless of the complexity and purposes. EC2 is one of their main essential features, which stands for Amazon Elastic Compute Cloud, that provide virtual computers to clients for their personal use.

(Session, 2022), in the scope of this project, is a piece of object data that contains a minimum of necessary information about a client that are visiting a server in a client-server protocol. Normally, client will need to provide their credentials in order to fully use the service to the fullest of their capabilities. However, the action of providing credentials would be repeated many times, causing lots of problems, mainly in the security and inconvenience aspects. But by creating a data object called session, saved it to the server and only give the session ID back to the client, the truthiness of the client-server relationship is now maintained, as only authenticated and authorized clients can have the session ID, thus making them able to use the server.

3 SPECIFICATIONS & ARCHITECTURE

3.1 Specifications

This passage describes the purpose and functionalities of this whole project in a list of short descriptions. The application is built based on how an online banking application should behave; therefore, these requirements should be mandatory.

- Authentication: the user can only gain access to protected resources when they can answer this question: Are they in the database?
- Authorization: the user can only gain access and modify to protected resources only when they can answer this question: Are they allowed to do this?
- Login system: the user can perform register, login and logout operations to receive/remove a session ID, which will be used as a way for authentication and authorization to work
- Bank statement: the user can order and view a collection of data pieces that are mimic to represent how a real-world banking application would provide a list of data piece of financial transactions
- Perform a transaction: the user can create a financial transaction record by filling in all the required data to form a financial transaction that should behaved accordingly to how a digital monetary transaction would behave

3.2 Architecture

3.2.1 Session-based Authentication and Authorization

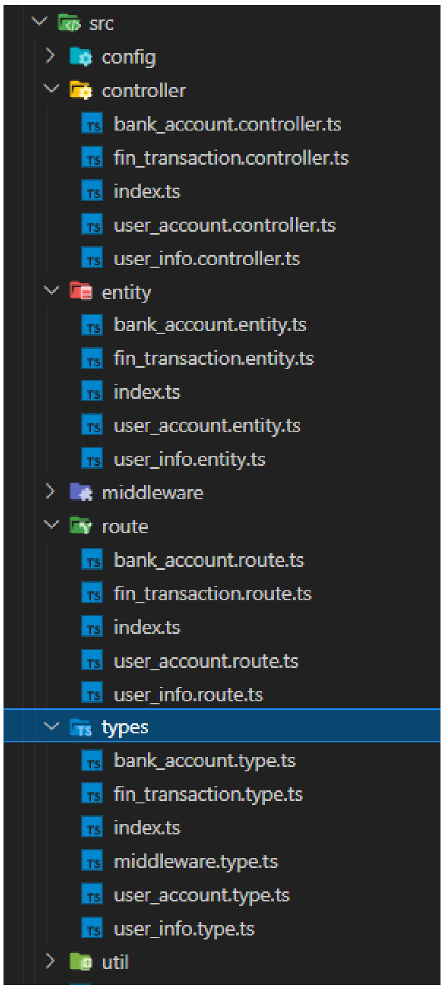
This project used session as a proof-of-identity to protect sensitive resources. However, another popular approach when building web application authentication and authorization would be the usage of JWT, a software development standard of how data object should be formed by combining optional encryption, required signature and the main payload; which contains the required information. While it is tempting to implement JWT for such a common use case in building web application, the session approach was chosen instead, for the following reasons:

- The session will be handled in the server, whereas JWT is handled in the client, making the process of handling proof-of-identity much simpler, considering the logic of the authentication and authorization is already server-sided.
- The session provides faster reaction times in revoking privileges than JWT: imagine a User' account being compromised. The server can easily mitigate this by destroying the session ID in question that containing the compromised user ID. This would be much more challenging in JWT case, as JWT is saved in the client, not the server's database
- The session suits better in the scope of the project. There is only one server maintaining all the resources, the authentication and the authorization, making many advantages JWT bring to the case wasted.

3.2.2 Features-based folder structure and N-layer architecture

Admittedly, this thesis is limited due to the author's knowledge. However, a mediocre implementation organized in a sound and battle-tested rules and regulations could still make the application easier to read, improve, maintain and fix.

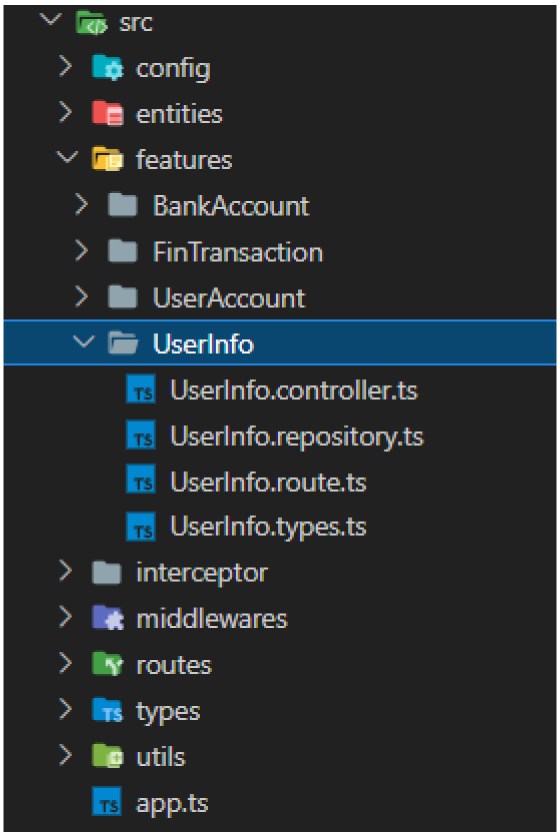
Regarding some of the earlier versions of the BE code, the code and logic were categorized based on their technical roles. However, there were troubles when maintaining the code or fixing a peculiar bug. Figure 9 gives an example of a bug presented in feature UInfo. The fix would be to jump around the code base and fix each related bugs on every single folder in the code base, making it much harder to track and visualize the code control flow.



```
src
  config
  controller
    bank_account.controller.ts
    fin_transaction.controller.ts
    index.ts
    user_account.controller.ts
    user_info.controller.ts
  entity
    bank_account.entity.ts
    fin_transaction.entity.ts
    index.ts
    user_account.entity.ts
    user_info.entity.ts
  middleware
  route
    bank_account.route.ts
    fin_transaction.route.ts
    index.ts
    user_account.route.ts
    user_info.route.ts
  types
    bank_account.type.ts
    fin_transaction.type.ts
    index.ts
    middleware.type.ts
    user_account.type.ts
    user_info.type.ts
  util
```

Figure 9. An earlier technical-based folder structure.

However, after some considerations and good insights, the approach of feature-based folder structure was seen a better choice to avoid the previously mentioned problems, while also dividing the whole BE application into nearly self-contained readable size chunks of code which helps lower the software complexity. In Figure 10, if an error was found in the feature UInfo, a revision and fixing would probably occurred only inside the UInfo components, as of now, nearly all features and attributes of the resource are being grouped together, making navigation between related files much easier.



```
src
  config
  entities
  features
    BankAccount
    FinTransaction
    UserAccount
    UserInfo
      UserInfo.controller.ts
      UserInfo.repository.ts
      UserInfo.route.ts
      UserInfo.types.ts
  interceptor
  middlewares
  routes
  types
  utils
  app.ts
```

Figure 10. Feature-based folder structure.

Another meaningful issue the author wishes to discuss together with the readers is the N-Layer Architecture. In software development, the N-Layer pattern is used to categorize the presentation, the application logic handling and data management functionalities into their own points of concern. Using this template, parts of the web application could be reusable, while developers could become more flexible and focus on each separate tasks at hand. The author tried to mimic the concepts, but errors and the inaccuracy understanding of the concepts are probably presented strongly in the thesis's work. He divided both the FE and the BE into three segregated blocks:

- The presentation layer: acts as the portal of interactions and interactions between the data consumers and what behinds the system (e.g., user interface in FE and the API routes in BE)
- The business layer: acts as the prime unit handling all the coordination, redirecting commands, produces logical decisions, calculations and processing data between the other two layers (e.g., controllers and middlewares in BE)
- The data access layer: acts as a direct communication between the application and the database to read and write information. (e.g., custom hooks with API folder in FE and entities with repositories in BE)

4 IMPLEMENTATION

This section presents the details of the implementation of the application. The following sections give explanations, the reason and concerns of the chosen approach, with pictures containing main snippets of the code that visualize the logic behind how each part was implemented.

4.1 Diagrams Design

Any good software should be started based on blueprints and templates making phase. The Unified Modeling Language (UML for short) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. Three diagrams are proposed in this section: the class diagram capturing how the static structure and classifiers of the system, the sequence diagram describing how interactions are produced when the Users use the Create Financial Transaction action, and the deployment diagram depicting how the thesis is maintained on an EC2 instance in Amazon Web Services.

The class diagram seen in Figure 11 depicts how all the features are interacting and relating to each other. This helps to speed up the writing of code, as there was a clear vision of how the web banking application should be built, and developers can reference Figure 11 when new features are materialized or modifications are needed.

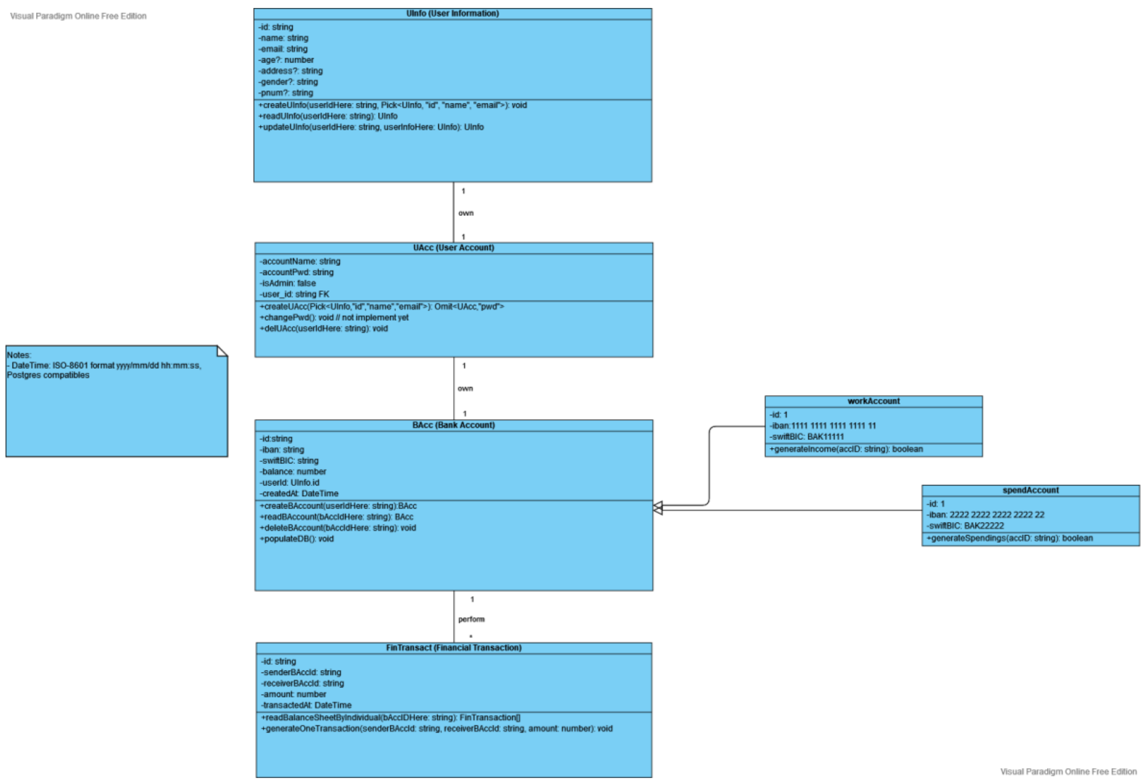

Visual Paradigm Online Free Edition
UInfo (User Information)
-id: string
-name: string
-email: string
-age?: number
-address?: string
-gender?: string
-pnum?: string
+createUInfo(userIdHere: string, Pick<UInfo, "id", "name", "email">): void
+readUInfo(userIdHere: string): UInfo
+updateUInfo(userIdHere: string, userInfoHere: UInfo): UInfo
1
own
1
UAcc (User Account)
-accountName: string
-accountPwd: string
-isAdmin: false
-user_id: string FK
+createUAcc(Pick<UInfo,"id","name","email">): Omit<UAcc,"pwd">
+changePwd(): void // not implement yet
+delUAcc(userIdHere: string): void
Notes:
- DateTime: ISO-8601 format yyyy/mm/dd hh:mm:ss, Postgres compatibles
1
own
1
BAcc (Bank Account)
-id:string
-iban: string
-swiftBIC: string
-balance: number
-userId: UInfo.id
-createdAt: DateTime
+createBAccount(userIdHere: string):BAcc
+readBAccount(bAccIdHere: string): BAcc
+deleteBAccount(bAccIdHere: string): void
+populateDB(): void
workAccount
-id: 1
-iban:1111 1111 1111 1111 11
-swiftBIC: BAK11111
+generateIncome(accID: string): boolean
spendAccount
-id: 1
-iban: 2222 2222 2222 2222 22
-swiftBIC: BAK22222
+generateSpendings(accID: string): boolean
1
perform
*
FinTransact (Financial Transaction)
-id: string
-senderBAccId: string
-receiverBAccId: string
-amount: number
-transactedAt: DateTime
+readBalanceSheetByIndividual(bAccIDHere: string): FinTransaction[]
+generateOneTransaction(senderBAccId: string, receiverBAccId: string, amount: number): void
Visual Paradigm Online Free Edition

Figure 11. Class diagram of the application.

One of the most important actions in this online banking web application is the action of creating a Financial Transaction. To further solidify the logic flow of how such an operation should be performed, Figure 12 depicts a sequence diagram. Basically, the Users must login first, to receive the user ID and the session ID from the backend. They would also receive a Cookie, which is a key-pair dictionary, containing the session ID inside. From thereafter, any requests during the duration of the session's expiration property, must contain a Cookie key in the header with the session ID as the value. Only then would the backend mark the authentication and authorization as passable and proceed further. The User will then do the Create a Financial Transaction action with the request body sending to the server containing the amount property, the sender and the receiver bank account ID. If everything so smoothly, the server will return the data of that newly created Fin-Transaction record, and the User could see it later in the bank account statement.

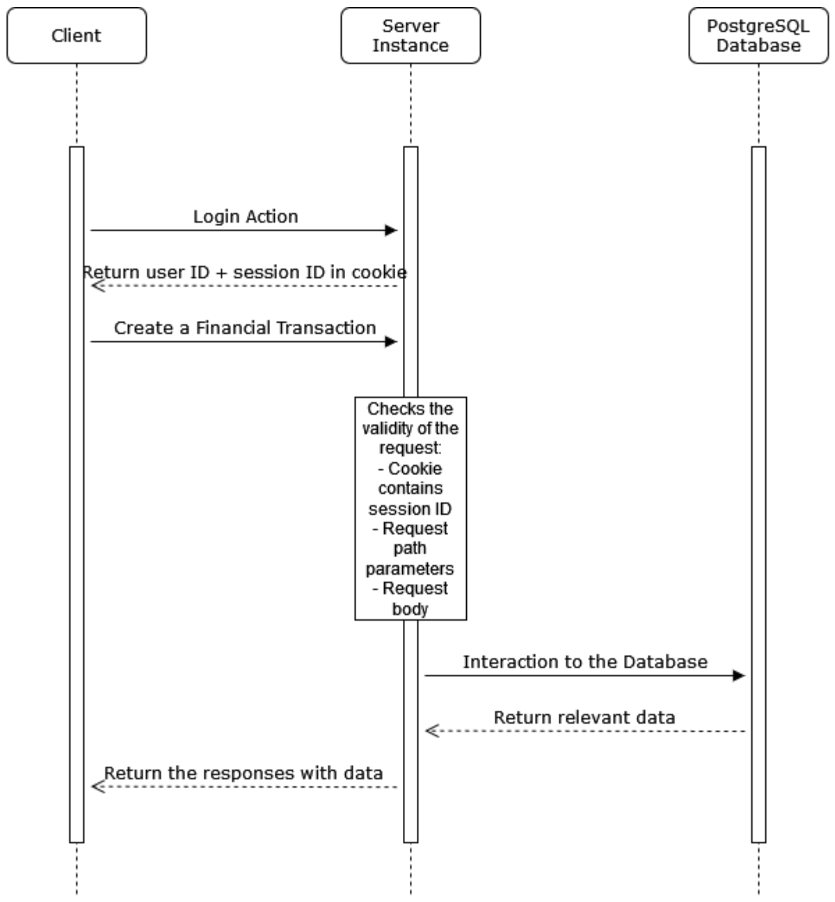

Client
Server Instance
PostgreSQL Database
Login Action
Return user ID + session ID in cookie
Create a Financial Transaction
Checks the validity of the request:
- Cookie contains session ID
- Request path parameters
- Request body
Interaction to the Database
Return relevant data
Return the responses with data

Figure 12. Sequence diagram of the application.

One of the final steps of finishing the project is the manual deployment to a virtual computer in AWS. The EC2 services were used with dynamic IP provided by Amazon to serve his application online. In Figure 13 shows how, the deployment was implemented.

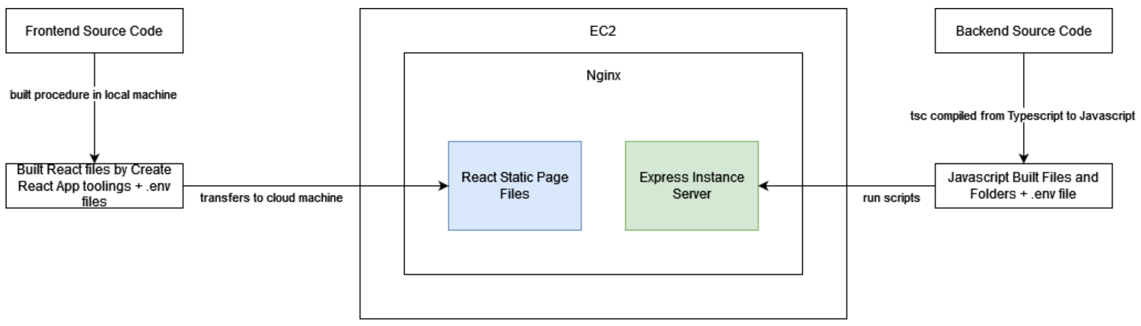
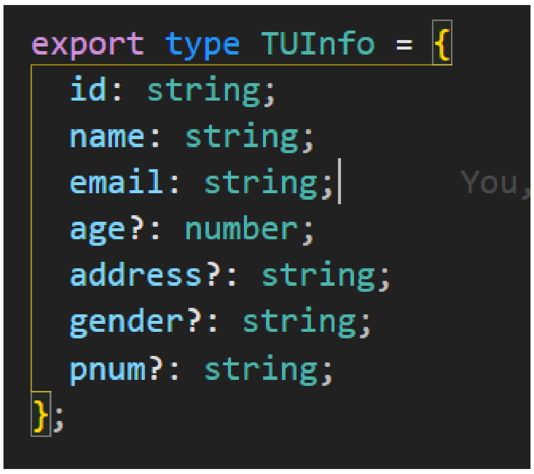

Frontend Source Code
built procedure in local machine
Built React files by Create React App toolings + .env files
transfers to cloud machine
EC2
Nginx
React Static Page Files
Express Instance Server
run scripts
Backend Source Code
tsc compiled from Typescript to Javascript
Javascript Built Files and Folders + .env file

Figure 13. Deployment diagram of the application.

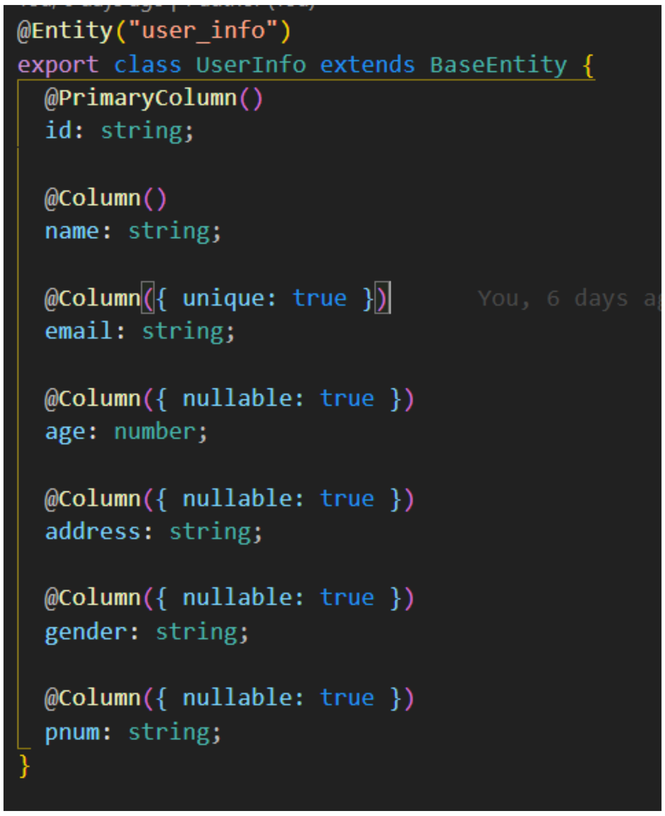
4.2 Database design

As previously mentioned in the Caveats, there are four main features in this thesis's work: the Bank Account feature (BAcc), the Financial Transaction feature (FinTransaction), the User Account feature (UAcc) and the User Information feature (UInfo). Being written using NodeJS and Express, the server would use PostgreSQL as the relational database of choices, while using TypeORM as the data access layer, acting as the communicator between the Express server instance and the database instance. Following the official instructions at TypeORM, a database driver is required (this project used pg as it was recommended also in the documentation when using PostgreSQL with TypeORM). Being a relational database, the data will be stored as relation models, organized in their own relevant tables of columns and rows, possibly with a unique identifier alongside. Below are the detail figures depicting how each feature should be constructed, formed and related to each other (two figures for each feature, one for the typings used internally by Typescript and one for the class mapping directly to the database table by TypeORM):



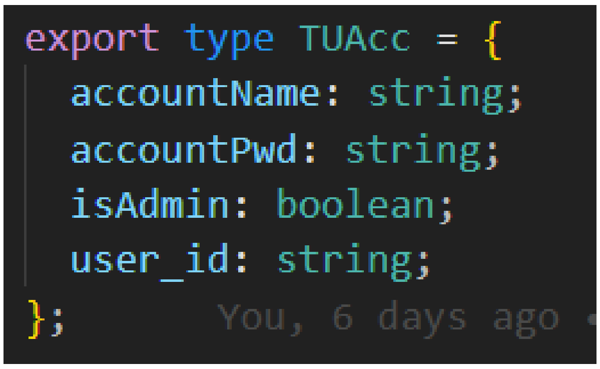
```
export type TUInfo = {
  id: string;
  name: string;
  email: string;        You,
  age?: number;
  address?: string;
  gender?: string;
  pnum?: string;
};
```

Figure 14. Typings for feature User Info.



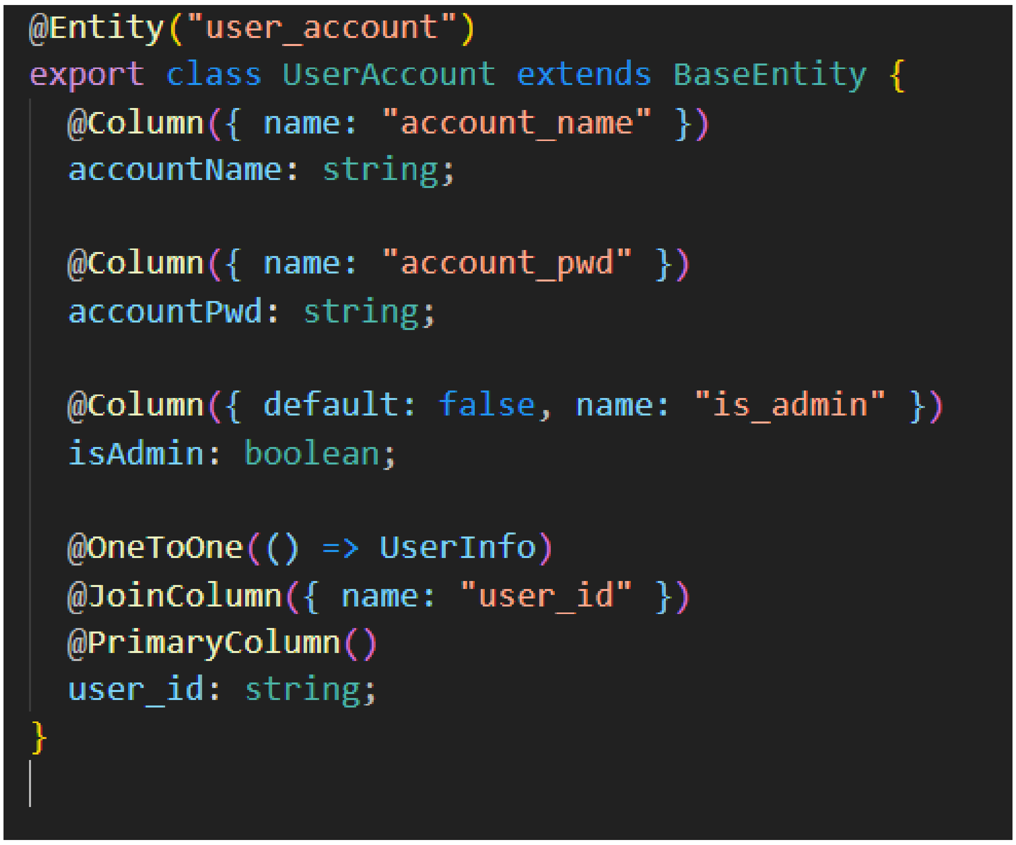
```
@Entity("user_info")
export class UserInfo extends BaseEntity {
  @PrimaryColumn()
  id: string;

  @Column()
  name: string;

  @Column({ unique: true })       You, 6 days a
  email: string;

  @Column({ nullable: true })
  age: number;

  @Column({ nullable: true })
  address: string;

  @Column({ nullable: true })
  gender: string;

  @Column({ nullable: true })
  pnum: string;
}
```

Figure 15. TypeORM entity class for feature User Info.



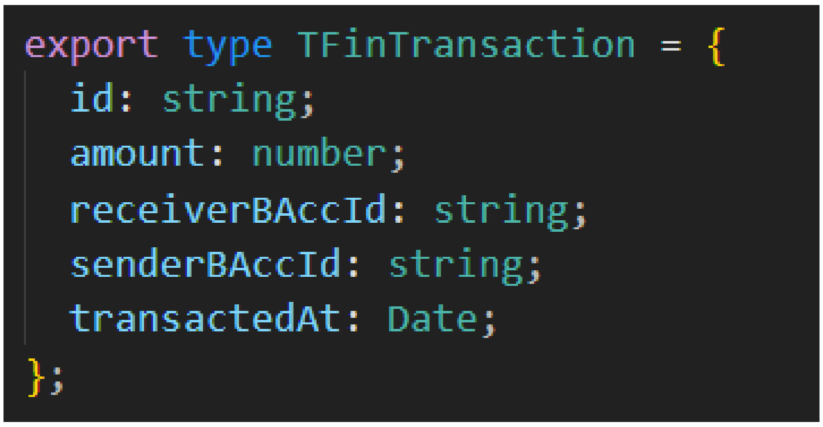
```
export type TUAcc = {
  accountName: string;
  accountPwd: string;
  isAdmin: boolean;
  user_id: string;
};        You, 6 days ago
```

Figure 16. Typings for feature User Account.



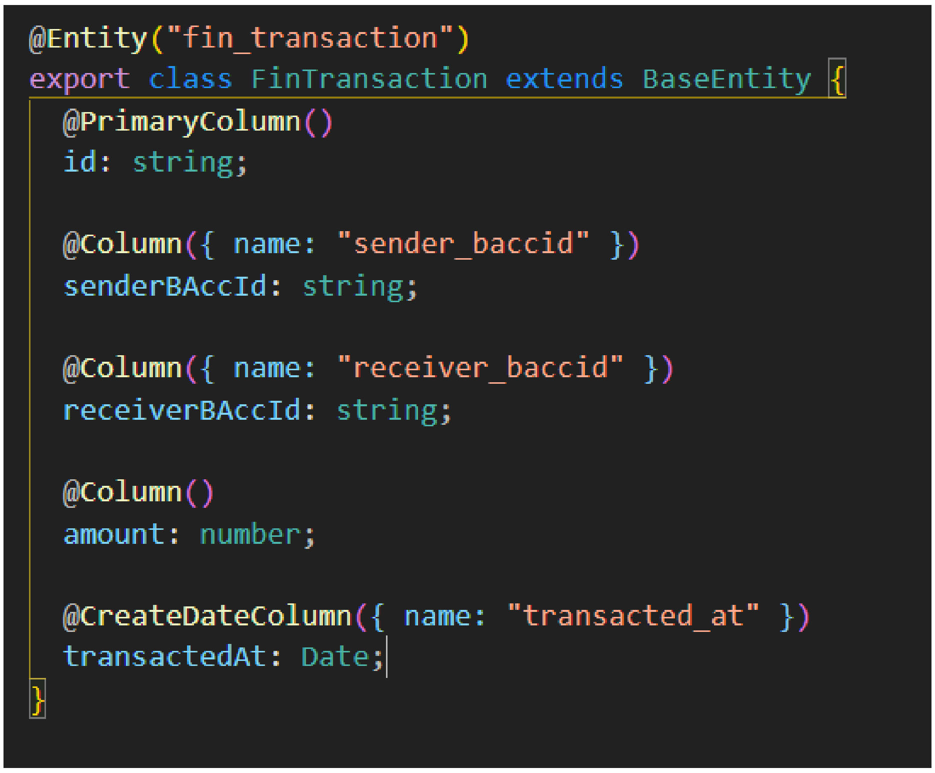
```
@Entity("user_account")
export class UserAccount extends BaseEntity {
  @Column({ name: "account_name" })
  accountName: string;

  @Column({ name: "account_pwd" })
  accountPwd: string;

  @Column({ default: false, name: "is_admin" })
  isAdmin: boolean;

  @OneToOne(() => UserInfo)
  @JoinColumn({ name: "user_id" })
  @PrimaryColumn()
  user_id: string;
}
```

Figure 17. TypeORM entity class for feature User Account.



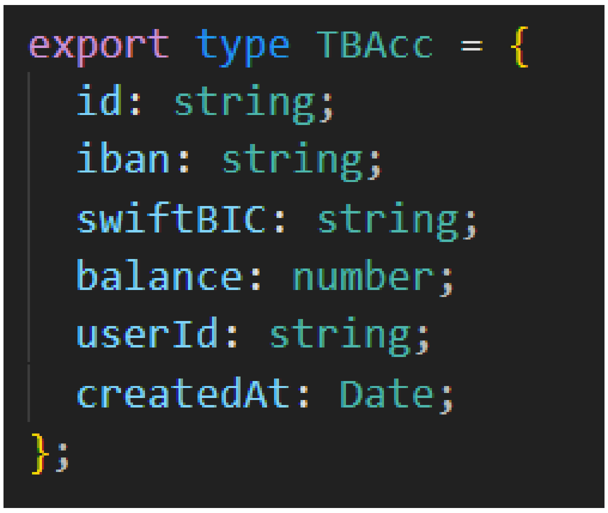
```
export type TFinTransaction = {
  id: string;
  amount: number;
  receiverBAccId: string;
  senderBAccId: string;
  transactedAt: Date;
};
```

Figure 18. Typings for feature Financial Transaction.



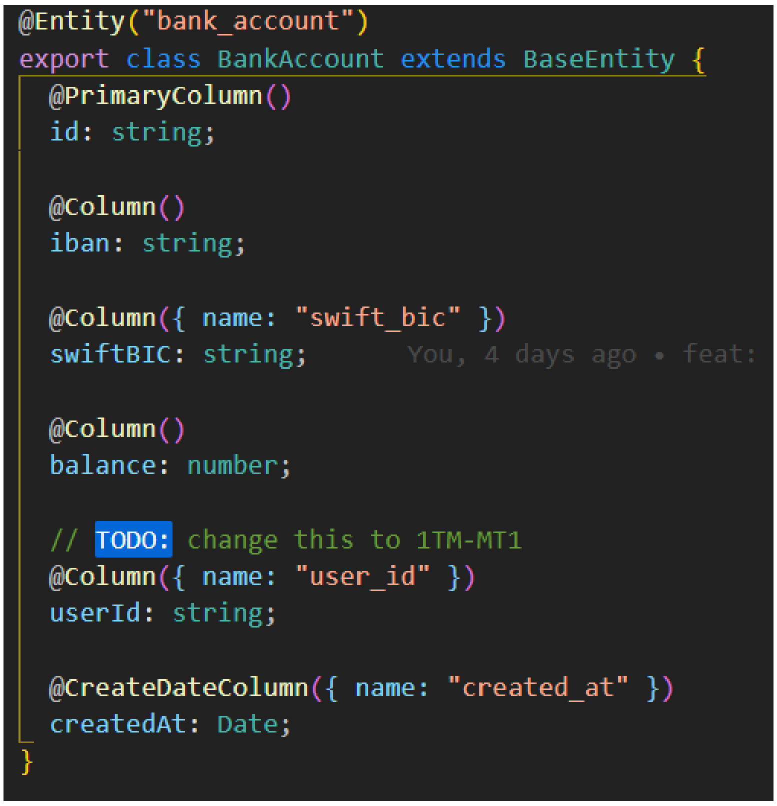
```
@Entity("fin_transaction")
export class FinTransaction extends BaseEntity {
  @PrimaryColumn()
  id: string;

  @Column({ name: "sender_baccid" })
  senderBAccId: string;

  @Column({ name: "receiver_baccid" })
  receiverBAccId: string;

  @Column()
  amount: number;

  @CreateDateColumn({ name: "transacted_at" })
  transactedAt: Date;
}
```

Figure 19. TypeORM entity class for feature Financial Transaction.



```
export type TBAcc = {
  id: string;
  iban: string;
  swiftBIC: string;
  balance: number;
  userId: string;
  createdAt: Date;
};
```

Figure 20. Typings for feature Bank Account.



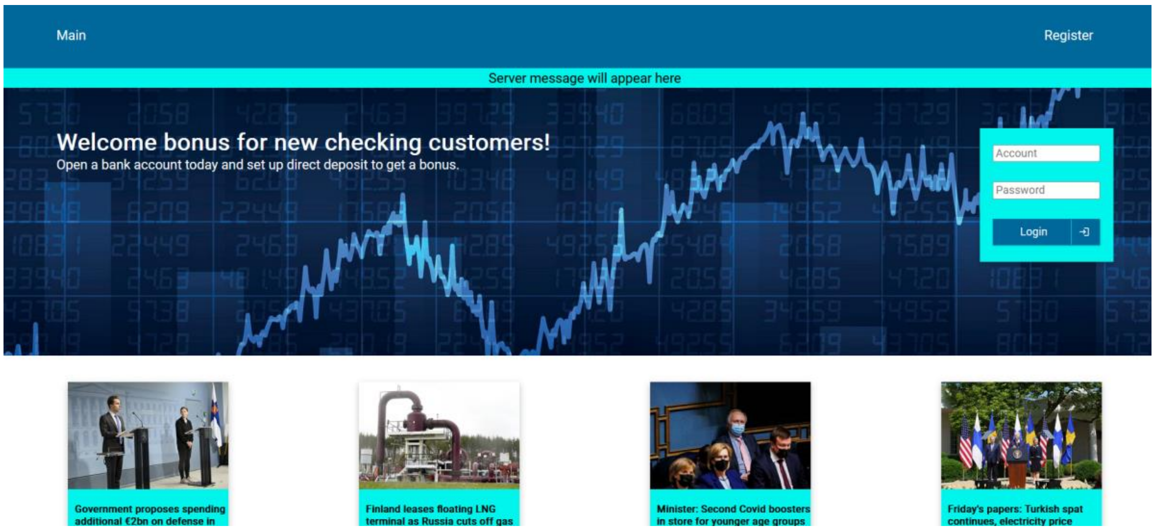
```
@Entity("bank_account")
export class BankAccount extends BaseEntity {
  @PrimaryColumn()
  id: string;

  @Column()
  iban: string;

  @Column({ name: "swift_bic" })
  swiftBIC: string;       You, 4 days ago • feat:

  @Column()
  balance: number;

  // TODO: change this to 1TM-MT1
  @Column({ name: "user_id" })
  userId: string;

  @CreateDateColumn({ name: "created_at" })
  createdAt: Date;
}
```

Figure 21. TypeORM entity class for feature Bank Account.

4.3 UI design

One objective of the thesis was to learn more about the world of UI/UX design. Using the approach of “reinventing the wheels”, which may be tedious, helps in understanding the core principles and concepts of how UI are drafted, built and combined together between HTML elements. Thus, the front-end of the thesis could be considered a weak point that needed much improvements, as styled-components were only used as assistance in building the design of the online banking application.

When visiting the website, the user first sees the Main Page. This page acts as a newsletter that contains all the links and anchors elements that are relating to the subject of the web application. This page, considering the real-life examples, contains information related to the banks that own the website, while it also offers some insights that are financially beneficial to the bank and the customers, such as promotions, deals, contracts, loans and saving information. While all the links in this page would be ghost links acting for demonstration only, please note the header that contains all the important anchors that will helps user navigate throughout the whole application.



Server message will appear here

Welcome bonus for new checking customers!
Open a bank account today and set up direct deposit to get a bonus.

Government proposes spending additional €2bn on defense in

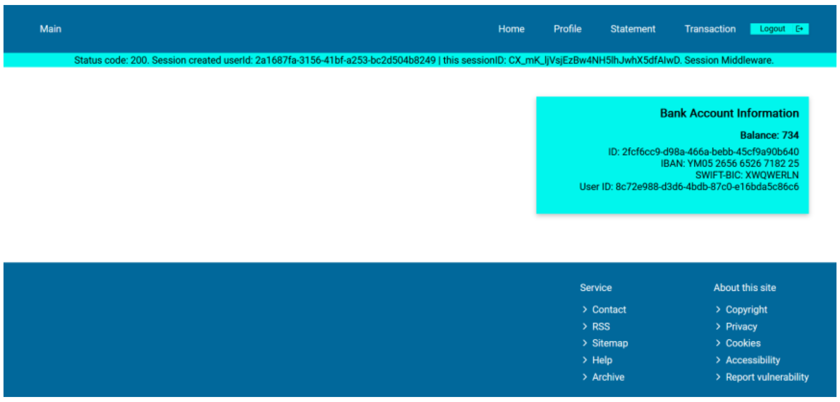
Finland leases floating LNG terminal as Russia cuts off gas

Minister: Second Covid boosters in store for younger age groups

Friday's papers: Turkish spat continues, electricity price

Figure 22. Mock up design of the Main Page

By clicking the Register link in the header, the User will navigate to the Register Page. The design of this page is kept simple, so the User sees a form with some input fields and many elements that are associated to them to help the User to understand which input field is for what data they provided. By design, after done registering, the User should move to the Login Page by clicking the Logic link in the header. Now, the User will also see a similar form like what they saw earlier in the Register Page. However, they only need to provide their user account name and the user account password in order to perform the login operation.



Status code: 200. Session created userId: 2a1687fa-3156-41bf-a253-bc2d504b8249 | this sessionID: CX_mK_ljVsjEzBw4NH5lhJwhX5dfAlwD. Session Middleware.

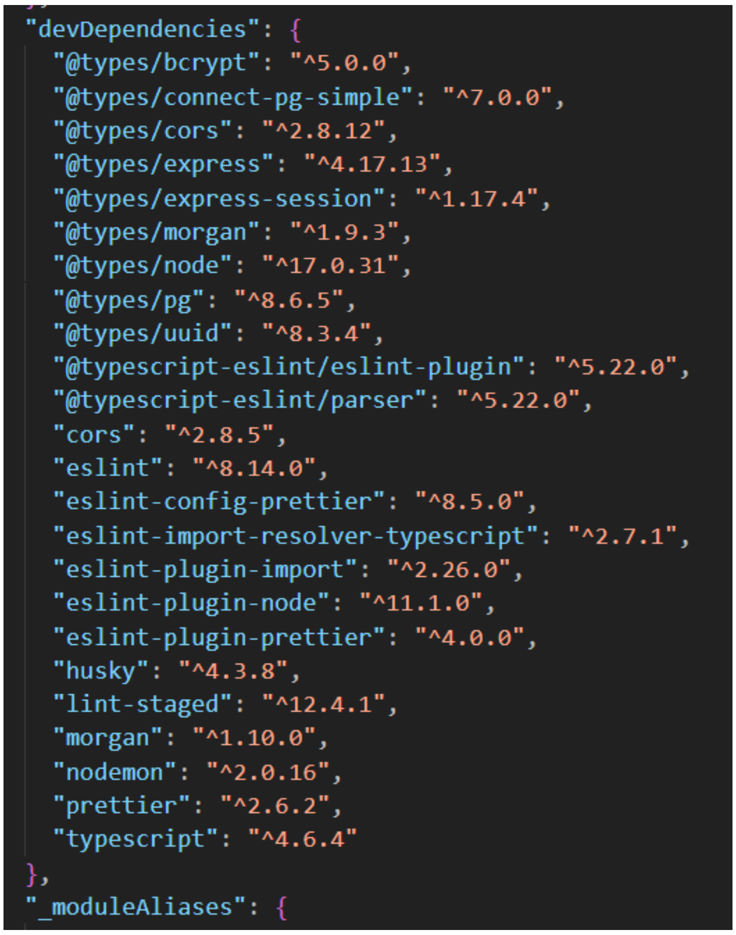
Bank Account Information
Balance: 734
ID: 2fcf6cc9-d98a-466a-bebb-45cf9a90b640
IBAN: YM05 2656 6526 7182 25
SWIFT-BIC: XWQWERLN
User ID: 8c72e988-d3d6-4bdb-87c0-e16bda5c86c6

Figure 23. Mock up design of protected User Home Page.

Hypothetically speaking, if the Users provide their credentials with no margin of errors, they would be introduced to the User Home Page. From here onwards, all the pages of the application will be protected. As in, only when the Users web browser contains the user ID and/or the session ID, would they be allowed to reach these pages. The Users should also note that the contents of the header have changed, as now there are four links corresponding to four services that this thesis' brings. The purpose this page is to provide the Users with a simple interface that contains all the information about a typical customer would use an online banking web application: a small section containing data about their bank account, some recent financial transaction they made and other minor matters that relate to all the features this web bank application provides. By default, every web application should have some kind of a Profile page or sorts. This application is not an exception. To view the personal data of the service User Information, the Users are advised to click the Profile link in the header, which will lead the Users to the Profile Page of the web app. In this page a small form is also presented that contains input fields of the all the non-volatile properties of the feature User Information that the Users can update.

4.4 Environment Setup

For the sake of simplicity and synchronousness, a similar developing environment was created which are used both in the FE and the BE with only minor modifications to serve its of their own purposes. Basically, each part of the thesis is handled by a repository tracked using Git, a popular version control system using "commits" to records changes made to the repository over time. The coding process is being done on the Visual Code Editor with helpful plugins, such as ENV, ESLint, Prettier, husky and lint-staged. All the libraries used to smooth and enhance the development experience can be seen in Figure 24. For more information, please visit their respective documentation online, as the explanation of those libraries are outside the scope of this thesis.

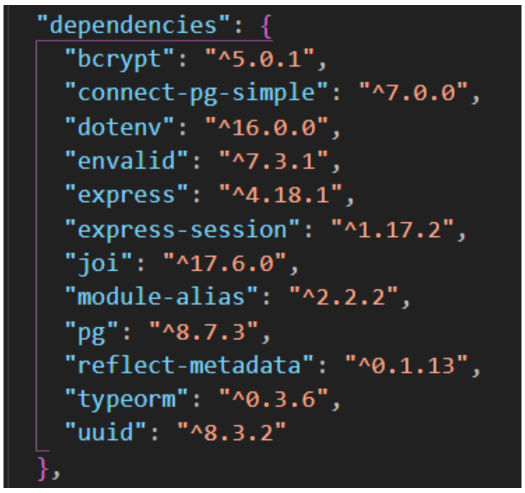


```
  "devDependencies": {
    "@types/bcrypt": "^5.0.0",
    "@types/connect-pg-simple": "^7.0.0",
    "@types/cors": "^2.8.12",
    "@types/express": "^4.17.13",
    "@types/express-session": "^1.17.4",
    "@types/morgan": "^1.9.3",
    "@types/node": "^17.0.31",
    "@types/pg": "^8.6.5",
    "@types/uuid": "^8.3.4",
    "@typescript-eslint/eslint-plugin": "^5.22.0",
    "@typescript-eslint/parser": "^5.22.0",
    "cors": "^2.8.5",
    "eslint": "^8.14.0",
    "eslint-config-prettier": "^8.5.0",
    "eslint-import-resolver-typescript": "^2.7.1",
    "eslint-plugin-import": "^2.26.0",
    "eslint-plugin-node": "^11.1.0",
    "eslint-plugin-prettier": "^4.0.0",
    "husky": "^4.3.8",
    "lint-staged": "^12.4.1",
    "morgan": "^1.10.0",
    "nodemon": "^2.0.16",
    "prettier": "^2.6.2",
    "typescript": "^4.6.4"
  },
  "_moduleAliases": {
```

Figure 24. Development environment setup.

4.5 Backend

The environment of the backend is a combination of the previous section and some specific JavaScript modules that are designed to help developing backend system. All libraries in Figure 25 are a must installation for the project to run properly.

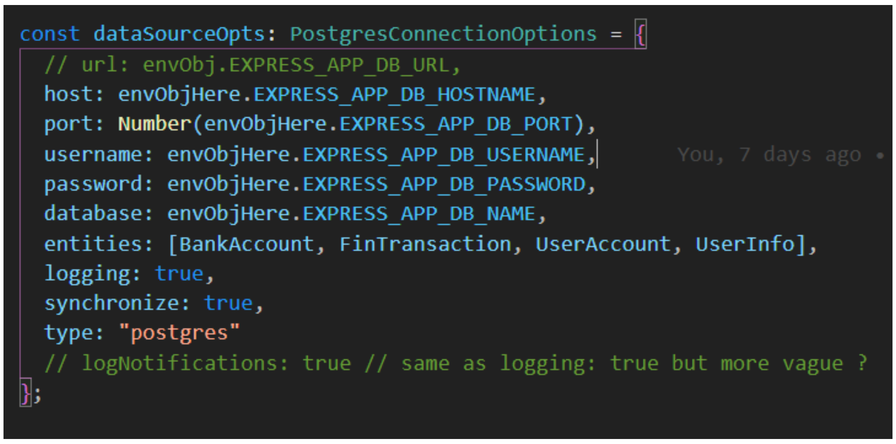


```
"dependencies": {
  "bcrypt": "^5.0.1",
  "connect-pg-simple": "^7.0.0",
  "dotenv": "^16.0.0",
  "envalid": "^7.3.1",
  "express": "^4.18.1",
  "express-session": "^1.17.2",
  "joi": "^17.6.0",
  "module-alias": "^2.2.2",
  "pg": "^8.7.3",
  "reflect-metadata": "^0.1.13",
  "typeorm": "^0.3.6",
  "uuid": "^8.3.2"
},
```

Figure 25. Required libraries for the back end.

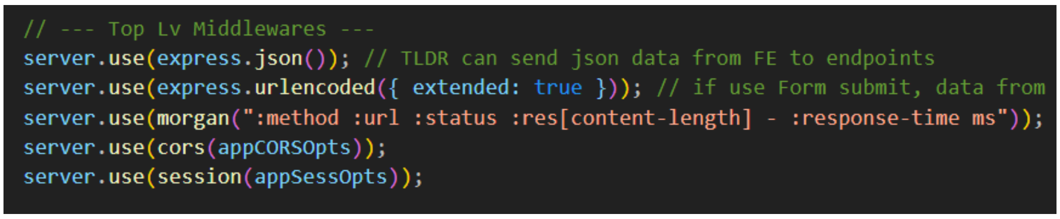
4.5.1 Configuration

The configuration folder contains the allocation of all the environment variables used in both the application and all the third-parties middleware. These can be wrapped as properties of an object and exported to the global scope for usage in other code segments. Below are examples of some third libraries used during the development.



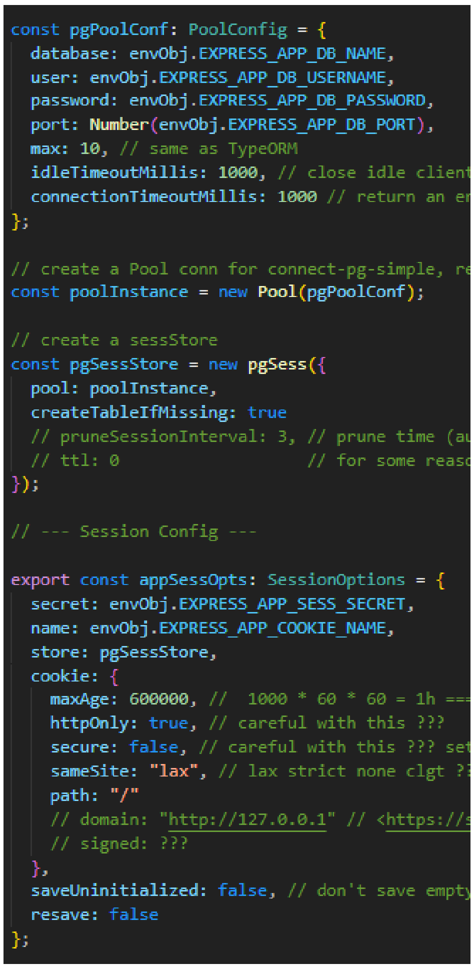
```
const dataSourceOpts: PostgresConnectionOptions = {
  // url: envObj.EXPRESS_APP_DB_URL,
  host: envObjHere.EXPRESS_APP_DB_HOSTNAME,
  port: Number(envObjHere.EXPRESS_APP_DB_PORT),
  username: envObjHere.EXPRESS_APP_DB_USERNAME,      You, 7 days ago •
  password: envObjHere.EXPRESS_APP_DB_PASSWORD,
  database: envObjHere.EXPRESS_APP_DB_NAME,
  entities: [BankAccount, FinTransaction, UserAccount, UserInfo],
  logging: true,
  synchronize: true,
  type: "postgres"
  // logNotifications: true // same as logging: true but more vague ?
};
```

Figure 26. A configuration snippet for TypeORM Data Source.



```
// --- Top Lv Middlewares ---
server.use(express.json()); // TLDR can send json data from FE to endpoints
server.use(express.urlencoded({ extended: true })); // if use Form submit, data from
server.use(morgan(":method :url :status :res[content-length] - :response-time ms"));
server.use(cors(appCORSOpts));
server.use(session(appSessOpts));
```

Figure 27. A configuration snippet for third-party middlewares.

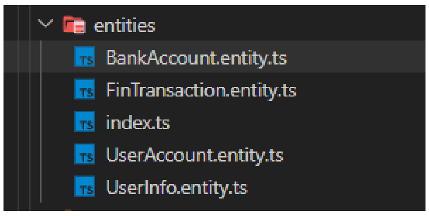


```
const pgPoolConf: PoolConfig = {
  database: envObj.EXPRESS_APP_DB_NAME,
  user: envObj.EXPRESS_APP_DB_USERNAME,
  password: envObj.EXPRESS_APP_DB_PASSWORD,
  port: Number(envObj.EXPRESS_APP_DB_PORT),
  max: 10, // same as TypeORM
  idleTimeoutMillis: 1000, // close idle client
  connectionTimeoutMillis: 1000 // return an er
};

// create a Pool conn for connect-pg-simple, re
const poolInstance = new Pool(pgPoolConf);

// create a sessStore
const pgSessStore = new pgSess({
  pool: poolInstance,
  createTableIfMissing: true
  // pruneSessionInterval: 3, // prune time (au
  // ttl: 0                   // for some reas
});

// --- Session Config ---

export const appSessOpts: SessionOptions = {
  secret: envObj.EXPRESS_APP_SESS_SECRET,
  name: envObj.EXPRESS_APP_COOKIE_NAME,
  store: pgSessStore,
  cookie: {
    maxAge: 600000, //  1000 * 60 * 60 = 1h ===
    httpOnly: true, // careful with this ???
    secure: false, // careful with this ??? set
    sameSite: "lax", // lax strict none clgt ??
    path: "/"
    // domain: "http://127.0.0.1" // <https://s
    // signed: ???
  },
  saveUninitialized: false, // don't save empty
  resave: false
};
```

Figure 28. A configuration snippet for express-session.

4.5.2 Entity

The Entity folder contains the blueprint of how each feature of the online banking application should be constructed. All features should be extended from the BaseEntity class to inherit the characteristics of a TypeORM entity while making easier to write code using the Repository's API provided by TypeORM. For more information, please refer to Database design section.

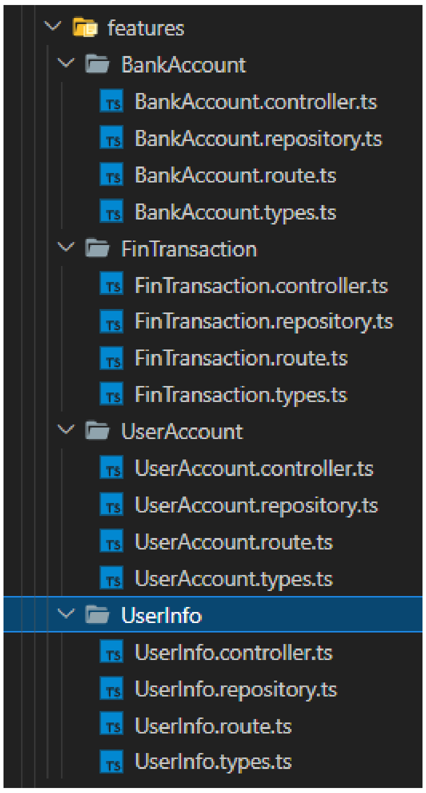


```
entities
  BankAccount.entity.ts
  FinTransaction.entity.ts
  index.ts
  UserAccount.entity.ts
  UserInfo.entity.ts
```

Figure 29. The location of the backend entities.

4.5.3 Features

Features are arguably the most important part of the whole thesis's backend. All the features of the web banking application are grouped together here. When the client sends request to the server, it will be routed to its designated features and will be proceeded under the controllers and the repository file. The controller part analyses the request and decide if it should pass it down to others middleware, such as the error handler. Otherwise, the request should be valid and be accepted. The repository file will jump it to interact with the database to mutate and modify data which would comply with the subject of the request. Finally, the controller would take data from the repository, create a response and send back to the client. Figure 30 shows how features are categorized in the application.



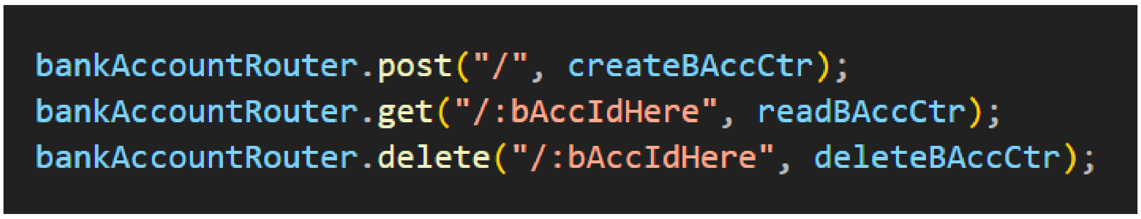
```
features
  BankAccount
    BankAccount.controller.ts
    BankAccount.repository.ts
    BankAccount.route.ts
    BankAccount.types.ts
  FinTransaction
    FinTransaction.controller.ts
    FinTransaction.repository.ts
    FinTransaction.route.ts
    FinTransaction.types.ts
  UserAccount
    UserAccount.controller.ts
    UserAccount.repository.ts
    UserAccount.route.ts
    UserAccount.types.ts
  UserInfo
    UserInfo.controller.ts
    UserInfo.repository.ts
    UserInfo.route.ts
    UserInfo.types.ts
```

Figure 30. The structure of the feature folder.

Due to size and length of code, only one feature is presented in detail: the Bank Account feature. Other features will be built using the same strategy mentioned in the above paragraph, with their own distinct logic handlings.

There are three public APIs that are connected to the Bank Account feature:

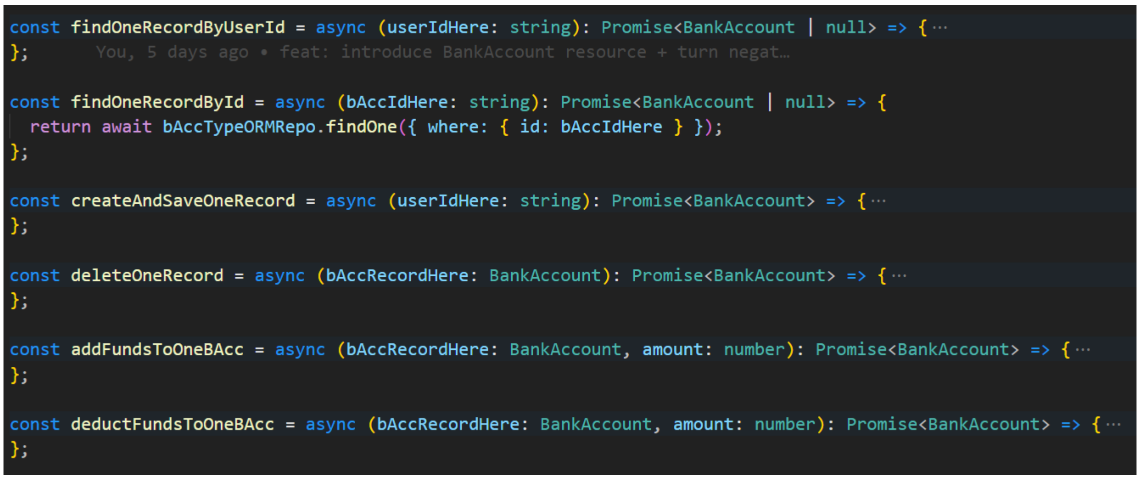
- The POST HTTP action to "/api/" to create a bank account
- The GET HTTP action to "/api/:bAccIdHere" to fetch data about a particular bank account, a bank account ID is required
- The DELETE HTTP action to "/api/:bAccIdHere" to delete the bank account, an ID is required to figure out what bank account record needed deleting



```
bankAccountRouter.post("/", createBAccCtr);
bankAccountRouter.get("/:bAccIdHere", readBAccCtr);
bankAccountRouter.delete("/:bAccIdHere", deleteBAccCtr);
```

Figure 31. The routing configuration of feature Bank Account.

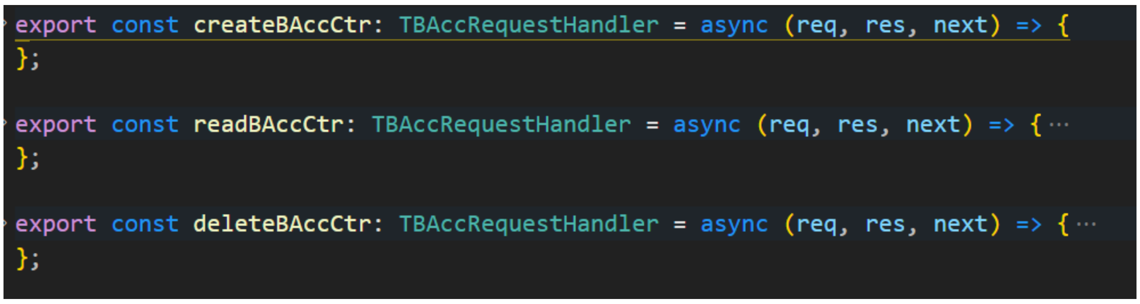
Each public API will be handled by appropriate controllers, whose names can be seen as the second arguments in all the action callers in Figure 31.



```
const findOneRecordByUserId = async (userIdHere: string): Promise<BankAccount | null> => {…
};      You, 5 days ago • feat: introduce BankAccount resource + turn negat…

const findOneRecordById = async (bAccIdHere: string): Promise<BankAccount | null> => {
  return await bAccTypeORMRepo.findOne({ where: { id: bAccIdHere } });
};

const createAndSaveOneRecord = async (userIdHere: string): Promise<BankAccount> => {…
};

const deleteOneRecord = async (bAccRecordHere: BankAccount): Promise<BankAccount> => {…
};

const addFundsToOneBAcc = async (bAccRecordHere: BankAccount, amount: number): Promise<BankAccount> => {…
};

const deductFundsToOneBAcc = async (bAccRecordHere: BankAccount, amount: number): Promise<BankAccount> => {…
};
```

Figure 32. Implementation of the repository of feature Bank Account.

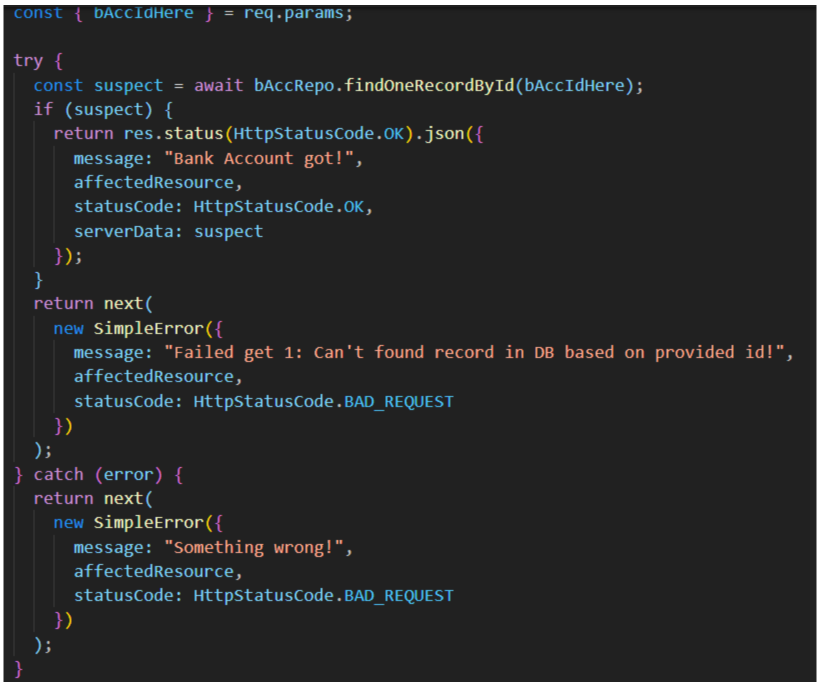
Figure 32 presents all the possible data acquisition and modifications available to the Bank Account resources in the database. Common CRUD operations can be seen at the top of the figure such as create, delete and read (through finding using ID or other properties). However, there are also unique operations that are limited in the feature Bank Account’s scope only, such as updating by changing properties of the bank account’s record such as balance. In the end, all operations will be wrapped inside a JavaScript object, ready to be exported to be used by the controllers.



```
export const createBAccCtr: TBAccRequestHandler = async (req, res, next) => {
};

export const readBAccCtr: TBAccRequestHandler = async (req, res, next) => {⋯
};

export const deleteBAccCtr: TBAccRequestHandler = async (req, res, next) => {⋯
};
```

Figure 33. Implementation of the controllers of feature Bank Account.

In Figure 33, the controller names can be seen, the same as in Figure 34. As mentioned above, each controller will handle its own specify API and use the exported repository to form the data, and send back the request if everything runs smoothly.



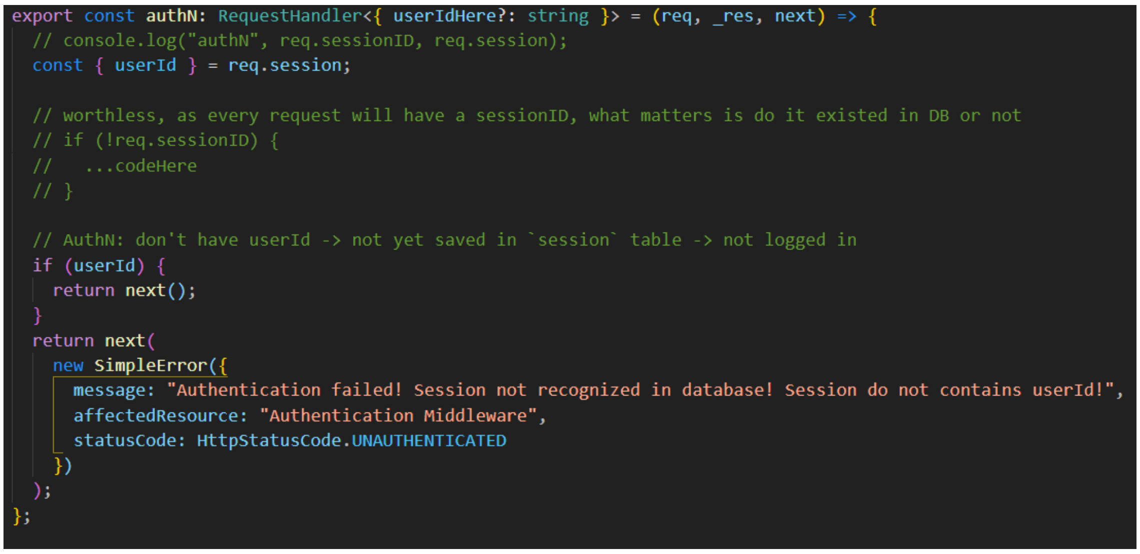
```
const { bAccIdHere } = req.params;

try {
  const suspect = await bAccRepo.findOneRecordById(bAccIdHere);
  if (suspect) {
    return res.status(HttpStatusCode.OK).json({
      message: "Bank Account got!",
      affectedResource,
      statusCode: HttpStatusCode.OK,
      serverData: suspect
    });
  }
  return next(
    new SimpleError({
      message: "Failed get 1: Can't found record in DB based on provided id!",
      affectedResource,
      statusCode: HttpStatusCode.BAD_REQUEST
    })
  );
} catch (error) {
  return next(
    new SimpleError({
      message: "Something wrong!",
      affectedResource,
      statusCode: HttpStatusCode.BAD_REQUEST
    })
  );
}
```

Figure 34. Implementation of a controller responses to a GET request.

This paragraph and Figure 34 will demonstrate how the logic in the controller was carried out, handling the fetching data operation of one bank account's record. At first, the path parameter must be extracted from the URL to get access to the ID of the bank account in question. If it does not exist, or being evaluated to a false value in JavaScript, the controller would “throw” an Error with details information of how it happened, then pass it down to the Error handling middleware for consumption. Trying to mimic the N Layer Architecture, the duty of the controller is only to only proceed data from the client’s request, produced an error data, pass it down to other middleware in chain, or return a proper response if everything is correct. The handling of errors and the validation of data will be performed by middleware before or after the chain of operations, resulting writing, reading and debugging code much simpler and rapidly.

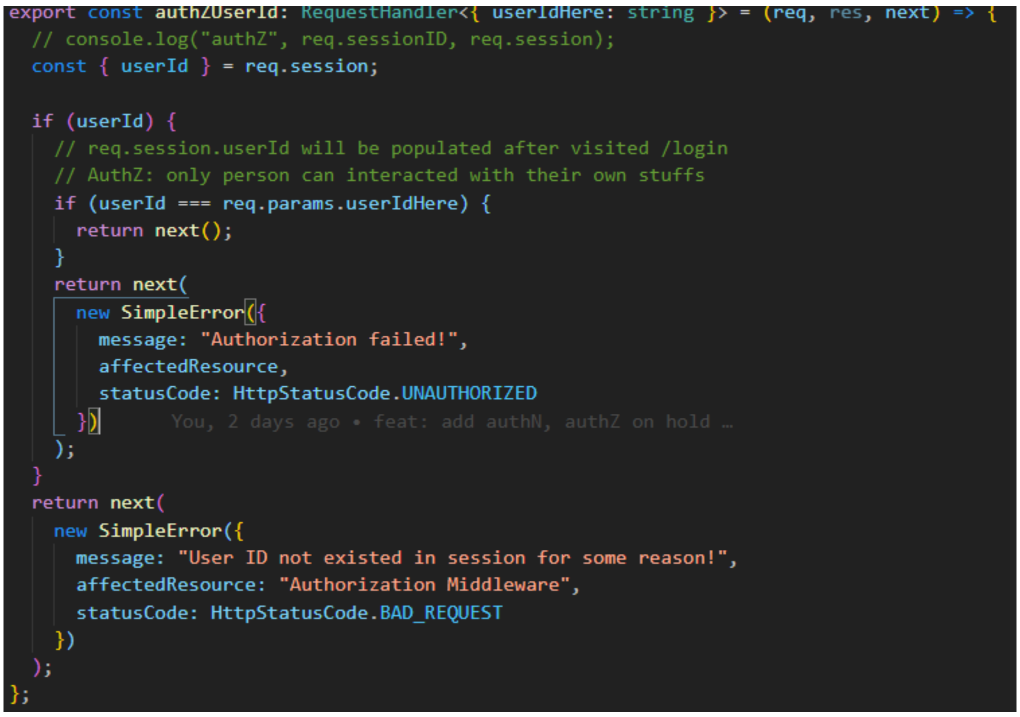
4.5.4 Application Middleware

In Express’s terminology, middleware is a function that has access to the data sent from the client, the data that will be sent back to the client and the forwarding logic to other functions if necessary. Any middleware that does not serve its purposes of handling requests and returning responses regarding to the core features of the project will be placed here, in the middleware folder. They are the authentication and authorization middleware, the session handler middleware, the validation middleware and many more.



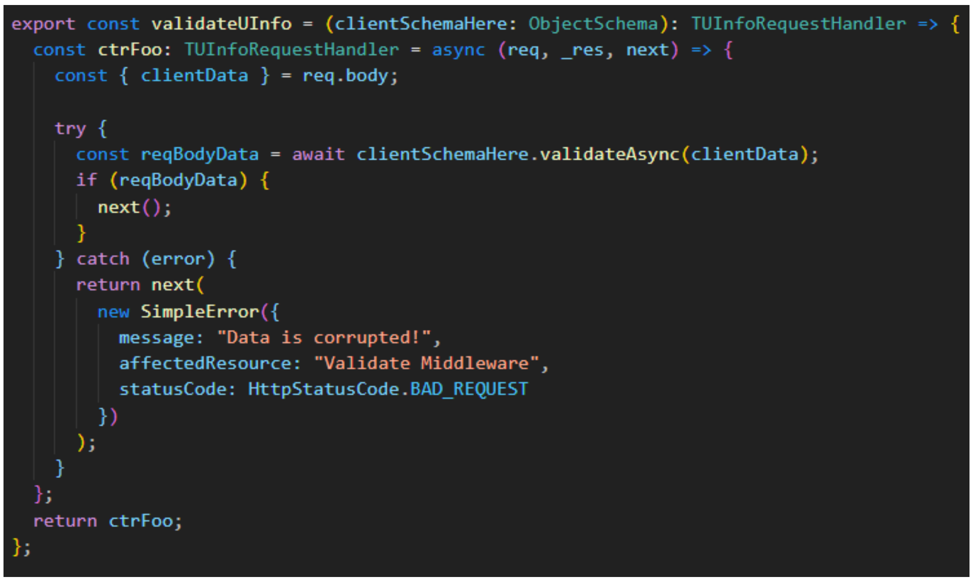
```
export const authN: RequestHandler<{ userIdHere?: string }> = (req, _res, next) => {
  // console.log("authN", req.sessionID, req.session);
  const { userId } = req.session;

  // worthless, as every request will have a sessionID, what matters is do it existed in DB or not
  // if (!req.sessionID) {
  //   ...codeHere
  // }

  // AuthN: don't have userId -> not yet saved in `session` table -> not logged in
  if (userId) {
    return next();
  }
  return next(
    new SimpleError({
      message: "Authentication failed! Session not recognized in database! Session do not contains userId!",
      affectedResource: "Authentication Middleware",
      statusCode: HttpStatusCode.UNAUTHENTICATED
    })
  );
};
```

Figure 35. Implementation of authentication logic.



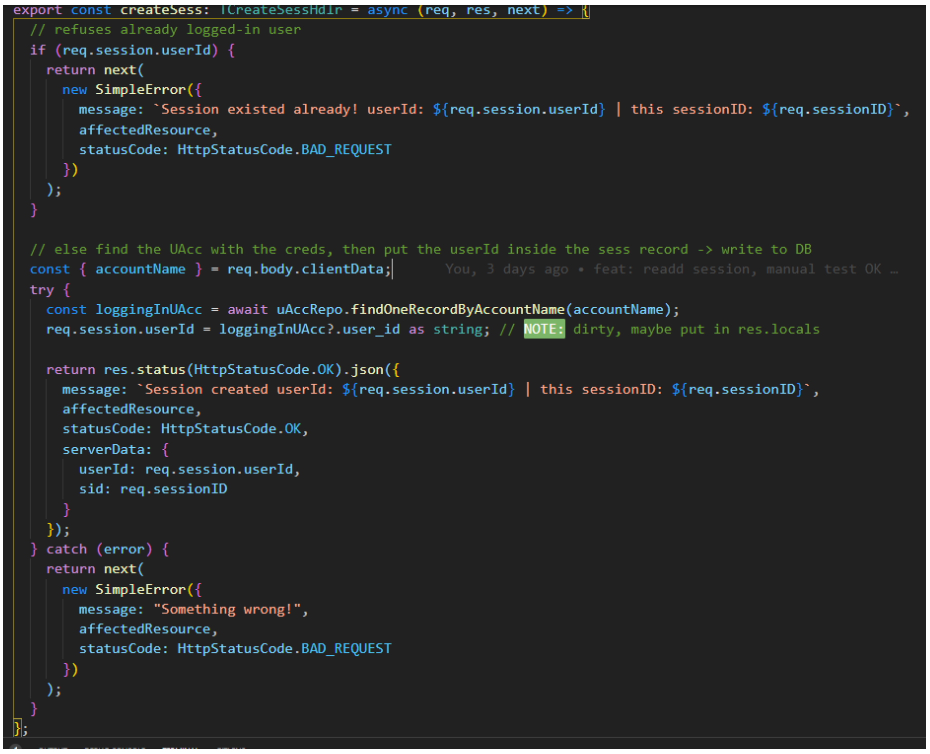
```
export const authZUserId: RequestHandler<{ userIdHere: string }> = (req, res, next) => {
  // console.log("authZ", req.sessionID, req.session);
  const { userId } = req.session;

  if (userId) {
    // req.session.userId will be populated after visited /login
    // AuthZ: only person can interacted with their own stuffs
    if (userId === req.params.userIdHere) {
      return next();
    }
    return next(
      new SimpleError({
        message: "Authorization failed!",
        affectedResource,
        statusCode: HttpStatusCode.UNAUTHORIZED
      })      You, 2 days ago • feat: add authN, authZ on hold …
    );
  }
  return next(
    new SimpleError({
      message: "User ID not existed in session for some reason!",
      affectedResource: "Authorization Middleware",
      statusCode: HttpStatusCode.BAD_REQUEST
    })
  );
};
```

Figure 36. Implementation of authorization logic.



```
export const validateUInfo = (clientSchemaHere: ObjectSchema): TUInfoRequestHandler => {
  const ctrFoo: TUInfoRequestHandler = async (req, _res, next) => {
    const { clientData } = req.body;

    try {
      const reqBodyData = await clientSchemaHere.validateAsync(clientData);
      if (reqBodyData) {
        next();
      }
    } catch (error) {
      return next(
        new SimpleError({
          message: "Data is corrupted!",
          affectedResource: "Validate Middleware",
          statusCode: HttpStatusCode.BAD_REQUEST
        })
      );
    }
  };
  return ctrFoo;
};
```

Figure 37. Implementation of validation logic.

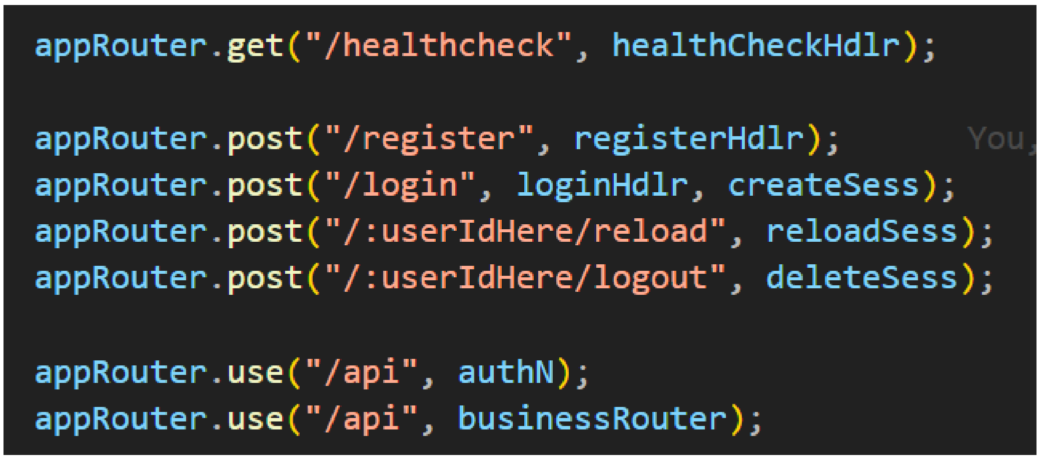


```
export const createSess: TCreateSessHdlr = async (req, res, next) => {
  // refuses already logged-in user
  if (req.session.userId) {
    return next(
      new SimpleError({
        message: `Session existed already! userId: ${req.session.userId} | this sessionID: ${req.sessionID}`,
        affectedResource,
        statusCode: HttpStatusCode.BAD_REQUEST
      })
    );
  }

  // else find the UAcc with the creds, then put the userId inside the sess record -> write to DB
  const { accountName } = req.body.clientData;       You, 3 days ago • feat: readd session, manual test OK …
  try {
    const loggingInUAcc = await uAccRepo.findOneRecordByAccountName(accountName);
    req.session.userId = loggingInUAcc?.user_id as string; // NOTE: dirty, maybe put in res.locals

    return res.status(HttpStatusCode.OK).json({
      message: `Session created userId: ${req.session.userId} | this sessionID: ${req.sessionID}`,
      affectedResource,
      statusCode: HttpStatusCode.OK,
      serverData: {
        userId: req.session.userId,
        sid: req.sessionID
      }
    });
  } catch (error) {
    return next(
      new SimpleError({
        message: "Something wrong!",
        affectedResource,
        statusCode: HttpStatusCode.BAD_REQUEST
      })
    );
  }
};
```

Figure 38. Partial snippet of session handling.

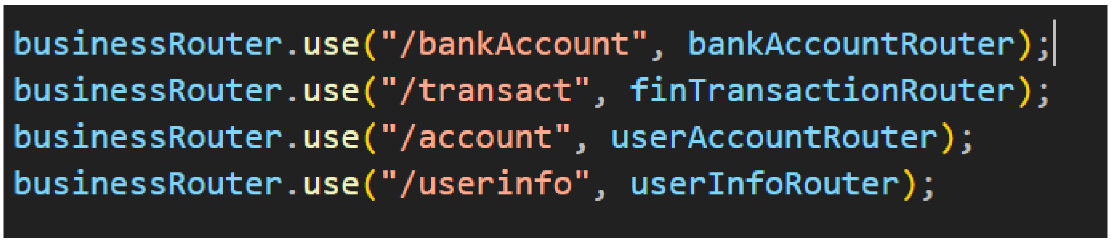
4.5.5 Routing

The routing logic was divided into two parts: the system parts and the business part. The system part will contain the code logic that handles common operations that can normally be seen other backend application: register, login, logout, authentication and authorization. The business part contains the code logic that handles specific operations that are only cater to this project and will probably not be available in other kinds of application such as creating an UAcc, create an UInfo, make a BAcc, or create a transaction.



```
appRouter.get("/healthcheck", healthCheckHdlr);

appRouter.post("/register", registerHdlr);        You,
appRouter.post("/login", loginHdlr, createSess);
appRouter.post("/:userIdHere/reload", reloadSess);
appRouter.post("/:userIdHere/logout", deleteSess);

appRouter.use("/api", authN);
appRouter.use("/api", businessRouter);
```

Figure 39. The app routing with the system routing on top.

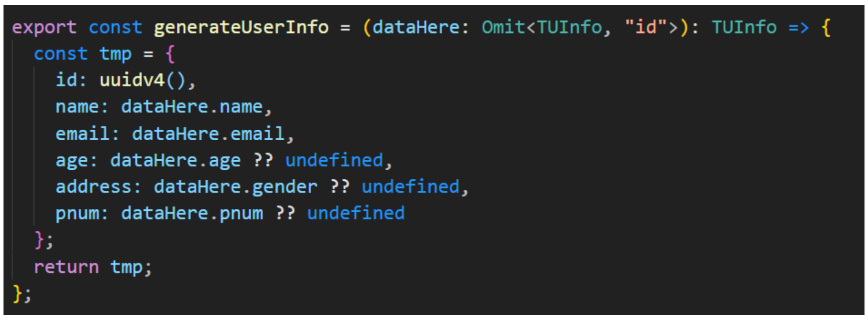


```
businessRouter.use("/bankAccount", bankAccountRouter);
businessRouter.use("/transact", finTransactionRouter);
businessRouter.use("/account", userAccountRouter);
businessRouter.use("/userinfo", userInfoRouter);
```

Figure 40. The business routing.

4.5.6 Utilities Code

The code that does not belong to a distinct point-of-concern will be grouped here. Usually, they are helper functions that compute and calculate data so that other sections of the code base do not need to rewrite themselves every time the same operation is needed. One of the most important aspects of this folder has been extracted below.



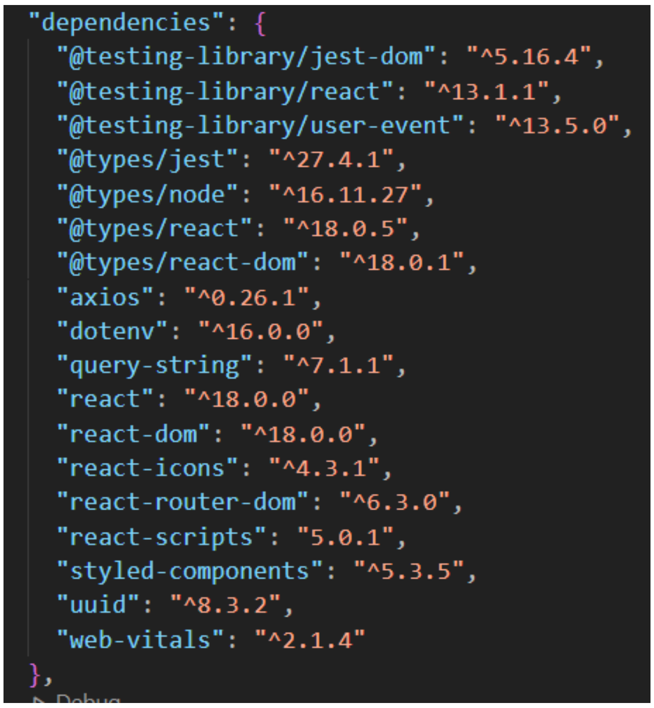
```
export const generateUserInfo = (dataHere: Omit<TUInfo, "id">): TUInfo => {
  const tmp = {
    id: uuidv4(),
    name: dataHere.name,
    email: dataHere.email,
    age: dataHere.age ?? undefined,
    address: dataHere.gender ?? undefined,
    pnum: dataHere.pnum ?? undefined
  };
  return tmp;
};
```

Figure 41. A helper snippet that generates feature User Info.

Note that all the other features have a similar kind of functions like this, but due to the code's length, the whole file is not posted here.

4.6 Frontend

Similar to the backend, the environment of the frontend is also set up in the way described in the Environment Setup section, while having some different libraries to help the development of the user interface. Figure 42 presents all the necessary JavaScript libraries are being used to build the front end of the application.

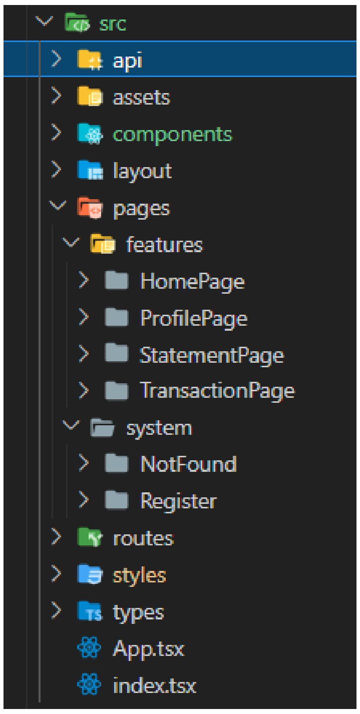


```
"dependencies": {
  "@testing-library/jest-dom": "^5.16.4",
  "@testing-library/react": "^13.1.1",
  "@testing-library/user-event": "^13.5.0",
  "@types/jest": "^27.4.1",
  "@types/node": "^16.11.27",
  "@types/react": "^18.0.5",
  "@types/react-dom": "^18.0.1",
  "axios": "^0.26.1",
  "dotenv": "^16.0.0",
  "query-string": "^7.1.1",
  "react": "^18.0.0",
  "react-dom": "^18.0.0",
  "react-icons": "^4.3.1",
  "react-router-dom": "^6.3.0",
  "react-scripts": "5.0.1",
  "styled-components": "^5.3.5",
  "uuid": "^8.3.2",
  "web-vitals": "^2.1.4"
},
```

Figure 42. Required libraries for the front end.

Acting as the data consumer, the frontend development was kept uncomplicated, to demonstrate the capabilities of the backend API. Figure 43 below depicts how the frontend is structured. The API folder contains code establishing communications to the backend, and returns the corresponding data for usage in frontend, regardless of failure or successful operations. The components folder, meanwhile, consists of many "building blocks" of the whole application, categorized in the frequency of reusing purpose. The layout folder holds code that directs how the order of those previously mentioned components, in this case will be wrapped between a footer and a header. The pages folder comprises all the core pages of the whole application. They are also divided based on N-Layer architecture to smoothen the developing experience. The routes folder handles the forwarding of path navigation when using the application. Regarding the styles and types folder, they will limit the code base to follow styling and code developing properly.

Finally, the main entry point of the application belongs to the app and the index files.



```
src
  api
  assets
  components
  layout
  pages
    features
      HomePage
      ProfilePage
      StatementPage
      TransactionPage
    system
      NotFound
      Register
  routes
  styles
  types
  App.tsx
  index.tsx
```

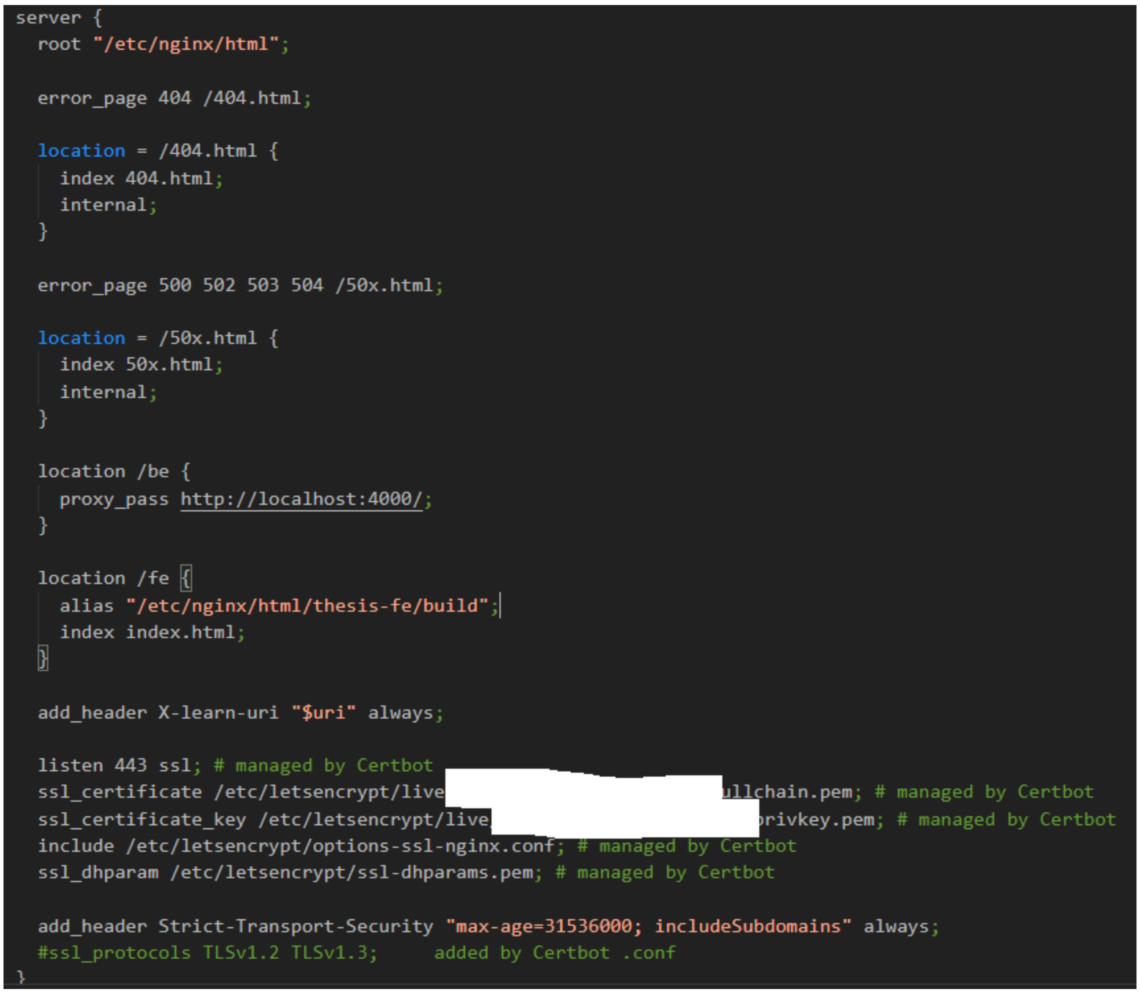
Figure 43. The folder structure of the frontend.

Due to complexity and length of the source code, detailed discussions of the implementation of the frontend will not be mentioned in this report. In short, the frontend is built with simplicity in mind, as the author emphasised his knowledge on the backend part of the thesis's work; while the frontend merely acts as the data consumption, the endpoint where data will be finalized and rendered. The Axios library is used to fetch data from the backend by encapsulated and extracted only the important parts, while also serve all HTTP requests with the credentials flag enabled to allow the usage of the session functionality provided from the backend. The website used Hooks and Context API to store required and sensitive data such as the session ID and the user ID to perform requests to the backend.

5 DEPLOYMENT AND TESTINGS

5.1 Deployment

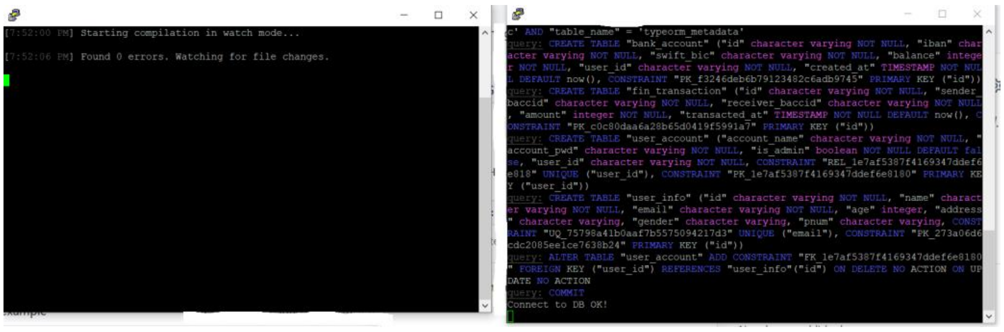
A Linux-based machine was created on the EC2 instance and Nginx set up as a proxy server to handle all the incoming internet traffic to the EC2 machine and to redirect them to appropriate processes, either the FE or the BE. Figure 44 demonstrates how the Nginx was configured.



```
server {
  root "/etc/nginx/html";

  error_page 404 /404.html;

  location = /404.html {
    index 404.html;
    internal;
  }

  error_page 500 502 503 504 /50x.html;

  location = /50x.html {
    index 50x.html;
    internal;
  }

  location /be {
    proxy_pass http://localhost:4000/;
  }

  location /fe {
    alias "/etc/nginx/html/thesis-fe/build";
    index index.html;
  }

  add_header X-learn-uri "$uri" always;

  listen 443 ssl; # managed by Certbot
  ssl_certificate /etc/letsencrypt/live[redacted]ullchain.pem; # managed by Certbot
  ssl_certificate_key /etc/letsencrypt/liverivkey.pem; # managed by Certbot
  include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
  ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

  add_header Strict-Transport-Security "max-age=31536000; includeSubdomains" always;
  #ssl_protocols TLSv1.2 TLSv1.3;     added by Certbot .conf
}
```

Figure 44. A configuration of Nginx hosting both frontend and backend.

Due to knowledge limitation, the whole application could only be deployed manually, instead of using automation scripts provided by Amazon command line or automation and deployment tooling such as Docker. A Windows user can send folders and files to a Linux machine by using WinSCP program, which uses the File Transfer Protocol (FTP) to transfer files and folders on a computer network. Next, the BE server process and the FE client process can be initiated by connecting to the EC2 bash windows through Putty with the private PGP key signature provided by Amazon when creating the EC2 instance.



```
[7:52:00 PM] Starting compilation in watch mode...

[7:52:06 PM] Found 0 errors. Watching for file changes.
```

```
c' AND "table_name" = 'typeorm_metadata'
query: CREATE TABLE "bank_account" ("id" character varying NOT NULL, "iban" char
acter varying NOT NULL, "swift_bic" character varying NOT NULL, "balance" intege
r NOT NULL, "user_id" character varying NOT NULL, "created_at" TIMESTAMP NOT NUL
L DEFAULT now(), CONSTRAINT "PK_f3246deb6b79123482c6adb9745" PRIMARY KEY ("id"))
query: CREATE TABLE "fin_transaction" ("id" character varying NOT NULL, "sender_
baccid" character varying NOT NULL, "receiver_baccid" character varying NOT NULL
, "amount" integer NOT NULL, "transacted_at" TIMESTAMP NOT NULL DEFAULT now(), C
ONSTRAINT "PK_c0c80daa6a28b65d0419f5991a7" PRIMARY KEY ("id"))
query: CREATE TABLE "user_account" ("account_name" character varying NOT NULL, "
account_pwd" character varying NOT NULL, "is_admin" boolean NOT NULL DEFAULT fal
se, "user_id" character varying NOT NULL, CONSTRAINT "REL_1e7af5387f4169347ddef6
e818" UNIQUE ("user_id"), CONSTRAINT "PK_1e7af5387f4169347ddef6e8180" PRIMARY KE
Y ("user_id"))
query: CREATE TABLE "user_info" ("id" character varying NOT NULL, "name" charact
er varying NOT NULL, "email" character varying NOT NULL, "age" integer, "address
" character varying, "gender" character varying, "pnum" character varying, CONST
RAINT "UQ_75798a41b0aaf7b5575094217d3" UNIQUE ("email"), CONSTRAINT "PK_273a06d6
cdc2085ee1ce7638b24" PRIMARY KEY ("id"))
query: ALTER TABLE "user_account" ADD CONSTRAINT "FK_1e7af5387f4169347ddef6e8180
" FOREIGN KEY ("user_id") REFERENCES "user_info"("id") ON DELETE NO ACTION ON UP
DATE NO ACTION
query: COMMIT
Connect to DB OK!
```

Figure 45. Built server code and running server instance.

The BE full source code can be pushed to the EC2 and being built and initialized there. However, the FE had be built on the author's own machine, as the EC2 instance are t2-micro, having limited capabilities and processing power, resulting in lagging and blocking behaviors when trying to produce a bundled, deployable React build files. As the time of writing, the application could only be accessible through a dynamic IPv4 address that was granted from the Amazon DNS. However, due to budget limitation, it may not be available in the foreseeable future.

5.2 Testing

Following recommended principles when developing applications, every software is required to be tested, regardless of the complexity or rigorousness of the test cases. Even a simple manual testing should suffice. Due to lack of experience, an automated testing script could not be successfully created. Instead, the idea of the API testing through Postman came up, an API platform with high usage in the developer community for consuming and testing backend API system. Figure 46 shows how the test case was structured:

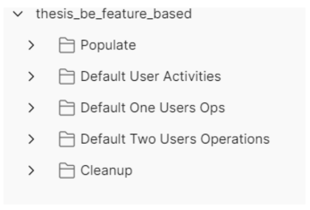
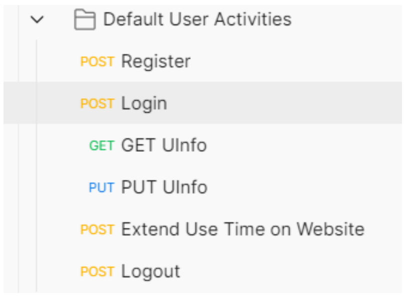


Figure 46. Structure of the API test suite.

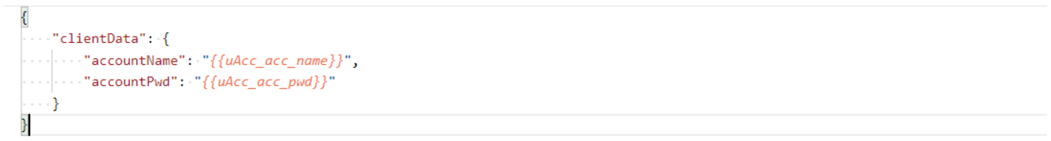
Before any operations, the application needed two bank account records acting as the “lender” and the “spender” so that all financial transactions could be traced and regulated as general concepts in trading. This operation was performed in Populate folder testing.

One of the first services that are mandatory to be tested in every software application is the registration, login and logout system. Figure 47 shows the testing files placed for previously mentioned services, while also testing the feature User Info action and the extension action on the session record. A detailed analysis on the Login action will be given in Figures 48, 49 and 50 respectively. After registration, the client will send a POST request, with the request’s body being a JSON object, as shown in Figure 48. Checking only the positive testing (aka all data inputs are valid), the server would return a user ID and the session ID in the response body. Furthermore, the client would also receive a Cookie containing the session ID which are referenced to a session record in the database, as depicted in Figure 49 and Figure 50.

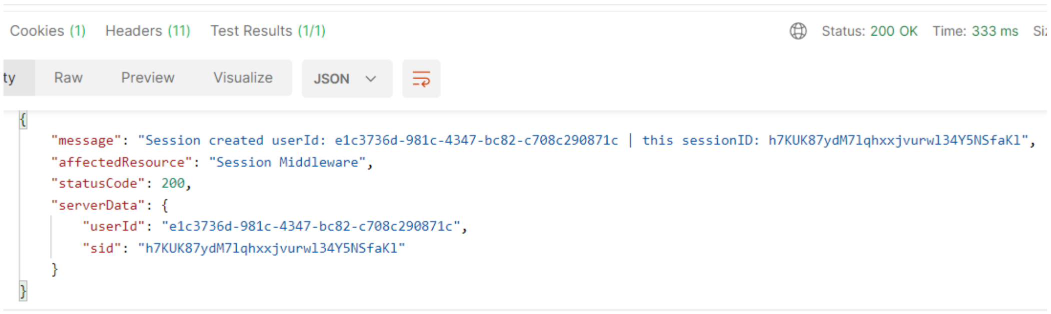


Default User Activities
POST Register
POST Login
GET GET UInfo
PUT PUT UInfo
POST Extend Use Time on Website
POST Logout

Figure 47. System testing files.

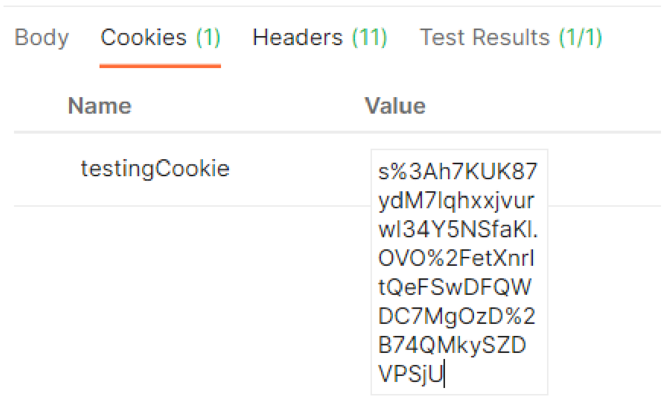


```
{
    "clientData": {
        "accountName": "{{uAcc_acc_name}}",
        "accountPwd": "{{uAcc_acc_pwd}}"
    }
}
```



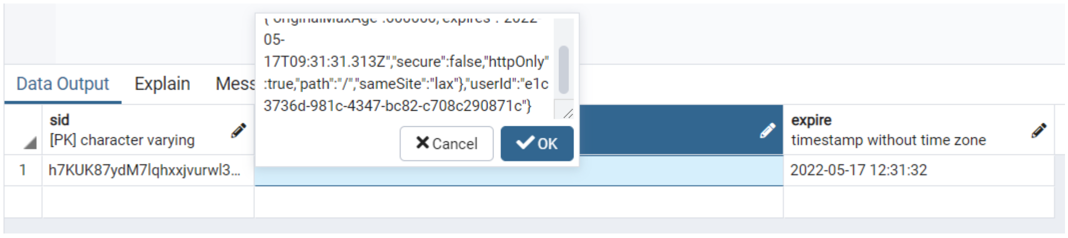
```
{
    "message": "Session created userId: e1c3736d-981c-4347-bc82-c708c290871c | this sessionID: h7KUK87ydM71qhxxjvurwl34Y5NSfaKl",
    "affectedResource": "Session Middleware",
    "statusCode": 200,
    "serverData": {
        "userId": "e1c3736d-981c-4347-bc82-c708c290871c",
        "sid": "h7KUK87ydM71qhxxjvurwl34Y5NSfaKl"
    }
}
```

Figure 48. Request and response examples of the Login action.



Name	Value
testingCookie	s%3Ah7KUK87ydM7lqhxxjvurwl34Y5NSfaKl.OVO%2FetXnrltQeFSwDFQWDC7MgOzD%2B74QMkySZDVPSjU

Figure 49. Client receives a Cookie with session ID.

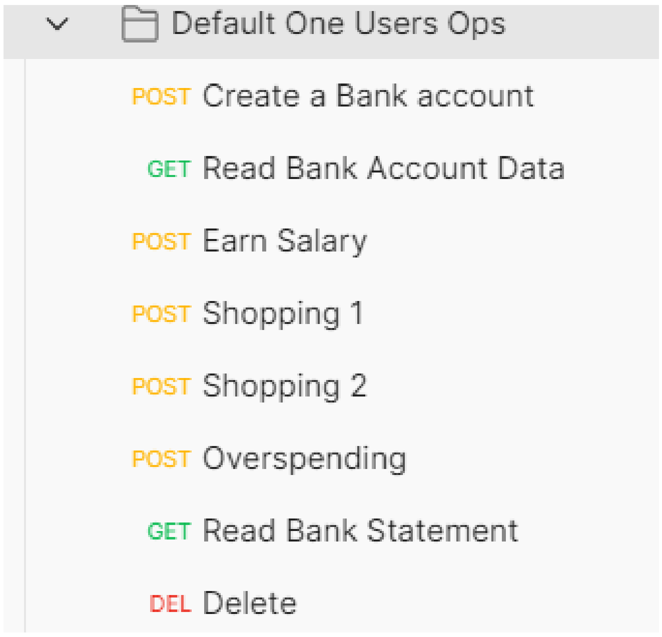


```
05-
17T09:31:31.313Z","secure":false,"httpOnly"
:true,"path":"/","sameSite":"lax"},"userId":"e1c
3736d-981c-4347-bc82-c708c290871c"}
```

	sid [PK] character varying		expire timestamp without time zone
1	h7KUK87ydM7lqhxxjvurwl3...		2022-05-17 12:31:32

Figure 50. The session record saved in the database.

The next two crucial testing folders are the One User default operations and Two Users default operations. They contain all the testing scripts for which actions that what a user would interact in such a website, and what two users would do also in the same website. Figure 51 and Figure 52 illustrate how the test for what actions was prepared.


Default One Users Ops

POST Create a Bank account

GET Read Bank Account Data

POST Earn Salary

POST Shopping 1

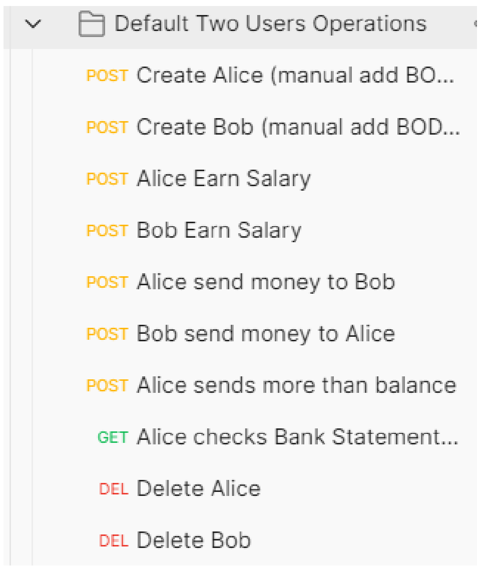
POST Shopping 2

POST Overspending

GET Read Bank Statement

DEL Delete

Figure 51. Typical one User operations testing case.


Default Two Users Operations

POST Create Alice (manual add BO...

POST Create Bob (manual add BOD...

POST Alice Earn Salary

POST Bob Earn Salary

POST Alice send money to Bob

POST Bob send money to Alice

POST Alice sends more than balance

GET Alice checks Bank Statement...

DEL Delete Alice

DEL Delete Bob

Figure 52. Typical two Users operations testing case.

A quick summary to explain what this test will behave: First, all requests must contain a session ID in the request headers to be able to continue. Then, data extracted from the request parameters and body would be evaluated if they are acceptable. Any failures occurring in the previous steps will fire the controllers to create a response with consequent errors. On the other hand, the business layer of the system would handle the requests, and output corresponding results. Those data from what the server returns will be used for rendering and displaying in the FE part of the project.

6 CONCLUSIONS

The thesis attempts to answer the question "How should an online banking web application be designed and an adequate illustration was developed of how a system should be researched and designed. However, during the research and development period, it became apparent there are much to be improved upon, such as the introduction to the credits systems, the multi-tier account features offered from real-world banking applications and the simulation system of trading stocks and bonds directly from the web app interface. Following that, the technical aspects and the coding paradigm of the project could be seen as immature, convoluted and imperfect if the code base are under reviews from an experience developer. To make marginal improvements on those factors, working experience is a must, which is lacking in this case. Having said that, comparing to what is being maintained and used in real life, the project has implemented most of the crucial requirements what should be expected from a banking web application: the management of a bank account and monetary transactional operations; combining with usual web application characteristics, such as session-based authentication and authorization and the login system. The project would still be developed after the evaluation as the author wishes to use it in his portfolio for future employment opportunities.

REFERENCES

MDN. Javascript. Accessed 15.05.2022. https://developer.mozilla.org/en-US/docs/Web/javascript

Typescript. Accessed 15.05.2022. https://www.typescriptlang.org/

React. Accessed 15.05.2022. https://reactjs.org/

React Router. Accessed 15.05.2022. https://reactrouter-dotcom.fly.dev/docs/en/v6

Styled Components. Accessed 15.05.2022. https://styled-components.com/

NodeJS. Accessed 15.05.2022. https://nodejs.org/en/

Express. Accessed 15.05.2022. http://expressjs.com/

Nginx. Accessed 15.05.2022. https://nginx.org/en/

PostgreSQL. Accessed 15.05.2022. https://www.postgresql.org/

TypeORM. Accessed 15.05.2022. https://typeorm.io/

AWS. Accessed 15.05.2022. https://aws.amazon.com/

Financial Transaction. Accessed 15.05.2022. https://en.wikipedia.org/wiki/Financial_transaction

NodeJS Best Practices. Accessed 15.05.2022. https://github.com/goldbergyoni/nodebestpractices

Stop using JWT for sessions. Accessed 15.05.2022. http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/

Stack Overflow. Authentication: JWT vs Sessions. Accessed 15.05.2022. https://stackoverflow.com/a/45214431/8834000

Wikipedia. Multitier Architecture. Accessed 15.05.2022. https://en.wikipedia.org/wiki/Multitier_architecture

APPENDICES

Acronyms & Abbreviations	Descriptions
E-bank/e-bank	Online banking web application.
Netbank	Same as above.
FE	Front-end, the user interface of a web application.
BE	Back-end, the data handle and access layer of a web application.
API	Application programming interface, where parts of programs can interact with each other to exchange data.
REST	Representational state transfer, a common software architecture for distributed systems.
CRUD	Create, read, update and delete; basic functions of a computer database.
SQL	Structured Query Language, a language to interact with data in a relational database management system.
AWS	Amazon Web Services, an integrated service by Amazon that provides many helpful features for developers.

EC2	Amazon Elastic Computer Cloud, a service from AWS that provide renting virtual computers.
ID	Identifier, used as a way to classify and determine each unique object, class, entity or records.
BAcc	Bank Account (BAcc for short): a data object that contains information that a typical bank account would normally have.
FinTransaction	Financial Transaction (FinTransaction for short): a data object that contains information of what a mon-etary transaction should legally have.
UAcc	User Account (UAcc for short): a data object that contains sensitive information about a user's creden-tials such as account name and password.
UInfo	User Information (UInfo for short): a data object that contains information about a typical user when using a website.