Course Name: PROGRAMMING FINANCE
Course Number and Section: **16:332:503**

FINAL PROJECT REPORT

B2C BANKING APPLICATION

Student Names:
Aditi Satish - 202003477 (fa402)
Khyati Patel - 202003448 (kp935)
Prachi Phatale - 199005660 (pp811)
Sudarshan Srinivasan - 182003539 (ss3020)

Date Submitted: 16 December 2021

Contribution Chart

Team Members	Percentage Allocation
Aditi Satish	**25%**
Khyati Patel	**25%**
Prachi Phatale	**25%**
Sudarshan Srinivasan	**25%**

Table of Contents:

Problem Statement

A project in the evolution of payment gateway from B2B business to B2C model. Creating a customer-centric banking application, which is exposed and marketed to the user. This application will offer REST-based APIs for integration with clients.

Glossary of Terms

1.REST API:

API, Application Programming Interface, is a set of protocols and definitions for building and integrating application software. API also works as a mediator between computers or systems. It helps to retrieve information or perform a function; it helps to communicate requirements from a particular system and fulfil the request. REST API, also known as RESTful API, is an application programming interface that fulfills the constraints of REST architectural style and allows interactions with RESTful web Services. REST stands for representational state transfer. It was created by computer scientist Roy Fielding. It consists of a set of resources which are interlinked. When a client makes his request via RESTful API, it transfers a representation of the state of the resource to the requester. There are several formats in which the request can be delivered. For e.g., HTTP: JSON (JavaScript Object Notation), HTML, XLT, Python. We used HTTP: JSON since it is language-agnostic, as well as readable by both humans and machines. If an API must be considered RESTful, it must fulfil some criteria. Though RESTful API has criteria to fulfil, it is still easier to use, faster, more lightweight, with increased scalability.

2.REACT:

React is a free and open-source front-end JavaScript library for building user interfaces based on UI components, [1]. React is written in JavaScript and is used to make better JavaScript applications. Hence, we refer to React specifically as a library. But what makes React a library and not a framework? The reason becomes clear when we look at other similar tools that are used to create complete web applications. For example: Angular, which shares the same purpose as React (to create single-page web apps). What sets it apart is the fact that when you set up an Angular project, it is bootstrapped with nearly everything that you will need to make a complete, large-scale app, but that is not the case with React. Since React is a library, the developer must choose the tools on their own [2].

3.NODE.JS

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripting—running scripts server-side

to produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language, rather than different languages for server-side and client-side scripts [3].

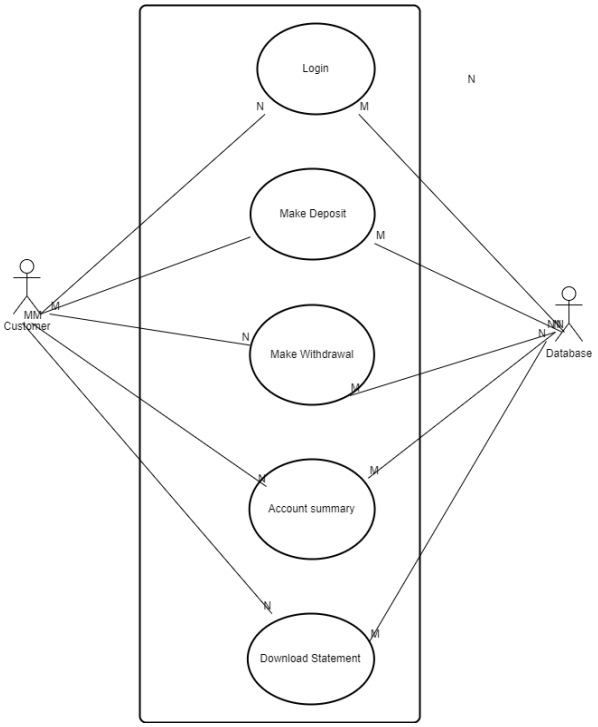
4. POSTMAN:

Postman is an application used for API testing. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain diverse types of responses that need to be subsequently validated.

5. POSTGRESQL

PostgreSQL is an advanced, enterprise-class, and open-source relational database system. PostgreSQL supports both SQL (relational) and JSON (non-relational) querying. It is a highly stable database backed by more than 20 years of development by the open-source community and is used as a primary database for many web applications as well as mobile and analytics applications. PostgreSQL has a rich history for support of advanced data types and supports a level of performance optimization that is common across its commercial database counterparts, like Oracle and SQL Server.

<u>Use Case Diagram:</u>



Algorithms/APIs used

Database

Common Use cases of PostgreSQL

The following are the common use cases of PostgreSQL.

1. A robust database in the LAPP stack: LAPP stands for Linux, Apache, PostgreSQL, and PHP (or Python and Perl). PostgreSQL is primarily used as a robust back-end database that powers many dynamic websites and web applications.
2. General-purpose transaction database: large corporations and start-ups alike use PostgreSQL as primary databases to support their applications and products.
3. Geospatial database: PostgreSQL with the PostGIS extension supports geospatial databases for geographic information systems (GIS).

PostgreSQL feature highlights

PostgreSQL has many advanced features that other enterprise-class database management systems offer, such as:

- User-defined types
- Table inheritance
- Sophisticated locking mechanism
- Foreign key referential integrity
- Views, rules, subquery
- Nested transactions (savepoints)
- Multi-version concurrency control (MVCC)
- Asynchronous replication
- Native Microsoft Windows Server version
- Tablespaces
- Point-in-time recovery

PostgreSQL is designed to be extensible, and it allows you to define your own data types, index types, functional languages, etc.

Benefits of using PostgreSQL

1. **Rich features and extensions:** PostgreSQL possesses robust feature sets including Multi-Version Concurrency Control (MVCC), point in time recovery, granular access controls, tablespaces, asynchronous replication, nested transactions, online/hot backups, a refined query planner/optimizer, and write-ahead logging. It supports international character sets, multi-byte character encodings, Unicode, and it is locale-aware for sorting, case sensitivity, and formatting. PostgreSQL is highly scalable both in the quantity of data it can manage and in the number of concurrent users it can accommodate.
2. **Reliability and standards compliance:** PostgreSQL's write-ahead logging makes it a highly fault-tolerant database. Its large base of open-source contributors lends it a built-in community support network. PostgreSQL is ACID compliant, and has full support for foreign keys, joins, views, triggers, and stored procedures, in many different languages. It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video.
3. **Open-source license:** PostgreSQL source code is available under an open-source license, granting you the freedom to use, modify, and implement it as you see fit, at no charge. PostgreSQL carries no licensing cost, which eliminates the risk for over-deployment. PostgreSQL's dedicated community of contributors and enthusiasts regularly find bugs and fixes, contributing to the overall security of the database system.

Why PostgreSQL over MySQL

1. **Database Performance:** Historically, MySQL has had a reputation as an extremely fast database for read-heavy workloads, sometimes at the cost of concurrency when mixed with write operations. PostgreSQL advertises itself as "the most advanced open-source relational database in the world". It was built to be feature-rich, extendable, and standards-compliant. In the past, Postgres performance was more balanced - reads were generally slower than MySQL, but it was capable of writing large amounts of data more efficiently, and it handled concurrency better.

   MySQL is still very fast at reading data, but only if using the old MyISAM engine. If using InnoDB (which allows transactions, key constraints, and other important features), differences are negligible (if they even exist). These features are absolutely critical to enterprise or consumer-scale applications, so using the old engine is not an option. On the other hand, MySQL has also been optimized to reduce the gap when it comes to heavy data writes.

   Postgres is an object-relational database, while MySQL is a purely relational database. This means that Postgres includes features like table inheritance and function overloading, which can be important to certain applications. Postgres also adheres more closely to SQL standards.

2. **Architecture:** PostgreSQL is an object-relational database, while MySQL is purely relational. This means PostgreSQL offers more complex data types and allows objects to inherit properties, but it also makes working with PostgreSQL more complex. PostgreSQL has a single, ACID-compliant storage engine. MySQL has support for 16 different storage engines suitable for different use cases. The default storage engine, InnoDB, provides index-organized tables.
   PostgreSQL also offers an INSTEAD OF trigger and can execute complex SQL statements in a trigger using functions. Both databases support standard SQL stored procedures, but PostgreSQL offers in addition the ability to call procedures written in languages other than SQL.

3. **Data types:** Postgres offers a wider variety of data types than MySQL. If the application deals with any of the unique data types it has available, or unstructured data, PostgreSQL may be a better pick. Like in our case, we were using a timestamp that is only present in PostgreSQL.

4. **Community:** Postgres claims it is 'the most advanced open-source database on the market', and this is not a hollow boast as many of its users second this bold assertion. Unlike MySQL, which is managed by Oracle, Postgres is overseen by a vibrant community of developers who are highly motivated to both use the product and help others to discover it and keep it running smoothly. Third-party support services are available, but because Postgres is community-owned, there is no commercial company backstopping it like MySQL with Oracle. However, the community is a highly engaged one and the same community supports and constantly updates the platform via the PostgreSQL Global Development Group.

Database creation for Banking transaction

We started with creating a database using CREATE DATABASE command, once the database was created, we used CREATE TABLE command to create tables for bank_user, TOKENS, account and transactions, our script.sql file is located inside the server folder, it has all the commands we used for creating tables and the rows inside the table.

For the bank_user, we are storing the user's first name, last name, email address, password and we are assigning user_id to each user, this acts as an identifier for that user. Email address and password is used in the login page in order to authenticate the user. In the token's table, we are assigning a unique token to each user who is doing some transaction. For the account table we are capturing the user's account number, bank name, IFSC Code(Routing number), total balance, along with things we are mapping user_id and account_id to each account number. Lastly, for the

transaction table we are capturing the user's transaction date, which is saved in timestamp, withdrawal amount, deposit amount, balance, and here we are mapping each transaction using account_id and transaction_id. Thus, using these ids make sure that each value is mapped properly to the respective entity and the transaction is captured properly.

Connecting database to Node.js

Next step is to connect the database to Node.js, in order to run the SQL scripts in the Node and save the data from those scripts. First step is to install "node-postgres" using the command 'npm install pg'. node-postgres is a pure JavaScript library that allows us to interact with a PostgreSQL database. It supports modern features such as async / await and is well maintained. node-postgres can be easily installed into the project by installing the pg package.

There are a couple of different ways to connect to the database. The node-postgres supports client and pool connections. A client is one static connection, and the pool manages the dynamic list of client objects with the automatic reconnection function.

We went with the pool option that can be used in case of having or expecting several simultaneous requests. Since our project is using the database frequently, using a single client connection to the database will likely slow down the application when we have many user requests. The easiest and most convenient way to address that problem was to use a connection pool. Usually, when a new client connects to the database, the process of establishing a connection and authenticating takes around 20-30 milliseconds. This is significant when we are running more queries leading to seconds of delay, which will likely end up being an unsatisfactory end-user experience.

Also, the PostgreSQL server can only handle a limited number of clients at a given time which will depend on your server memory. So, if 100 queries are going on in a second - this limitation might crash the server. Additionally, the client can process only one request at a time for a single connection which further slows things down.

Inside the server/db folder we created a file *connect.js* and added the following content in the file. We provided the details of our database; this will connect node to the postgres. The placeholders in the script below should be adjusted using the appropriate connection information (provided within an email for your PostgreSQL container):

- {user} — username to log into database with
- {password} — password for the specified user
- {host} — link to your PostgreSQL container
- {database} — database to be accessed (e.g., the default one — PostgreSQL)
- {port} — port number the database server listens to. The default one for PostgreSQL is 5432

```
const pool = new Pool({
user: 'postgres',
  password: 'Khyushu#123',
  host: 'localhost',
  port: 5432,
  database: 'b2c_payment'
});
const getClient = async () => {
  try {
    const client = await pool.connect();
    return client;
  } catch (error) {
    return null;
  }
};
 module.exports = { pool, getClient };
```

Login API Structure and Functionality

The login page for the b2c application was created using JavaScript, Node.js and Express. The login page is the first step in the creation of this application. Our application is built using a PostgreSQL database. Our application is built using the React library in JavaScript. First, a new project was created using the command “create-react-app". Create React App is a comfortable environment for learning React and is the best way to start building a new single-page application in React. When we use the create-react-app tool it creates a hierarchy of files and folder [5]. The two most significant folders which are created and are extensively used include the “src” and the “server” folder.

The SRC folder:

In simplest form it is our react app folder i.e., containing components, tests, CSS files etc. It is the mind of our app. Initially, we delete all the files from the src folder, and then create a new *index.js* file. This file renders our component and registers service workers (unregistered by default). Apart from the *index.js* file, we also create actions folder, components folder, css, reducers, router, store and utils folders inside the src folder. Actions folder contains all the actions of the application. For example, the index file in the actions folder exports all the actions that the application would dispatch from its components. The reducers folder contains the reducer files for every type of state.

After the creation of the folders, we create another *index.js* file inside the components folder. Note that the components folder is inside the src folder. Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions but work in isolation and return HTML. The *Register.js* file was also created inside the components folder with content that helped us in creation of the user registration form. Then we create a new file *constants.js* inside the utils folder for exporting Login and Logout buttons. The utils folder is a place where you can place small snippets that can be used throughout the application. Basically, small functions or components which can be used to build bigger components. A new file *common.js* inside the utils folder was also created, which is used for field validation. After the completion of this step, a new file called *“auth.js”* was created inside the reducers folder. The reducers folder contains all the Redux reducers for the project, each reducer updates a different part of the application state in response to dispatched redux actions. Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. After this step, we created a *store.js* file inside the /src/store folder. In this file, we have created a redux store with a single authentication reducer. We also added configuration for the redux dev tool to see Realtime actions dispatched.

Inside the /src/router folder, *AppRouter.js* file was created. This file is used for routing purposes. In React, routers help create and navigate between the different URLs that make up your web application. They allow your user to move between the components of your app while preserving user state and can provide unique URLs for these components to make them more shareable. With routers, you can improve your app's user experience by simplifying site navigation. Inside this file we use the "react-router-dom" library, which enables implementation of dynamic routing in a web app. The CSS folder inside the src folder contains the main CSS components of the web application. These components are stored inside a file called main.scss inside the CSS folder.

The Server Folder:

Using the above steps, we finished constructing the skeleton of our page. Moving forward, we navigate out of the src folder and go into the server folder. The server folder is used for server-side coding and testing in a local environment. Now, we create db, middleware, routes, utils and views folder inside the server folder along with an *index.js* file inside the main server folder. The "*index.js*" file is used to create and run the express server. It funnels incoming requests through a chain of middleware (of steps) where we can do something with the request, read some data from it, manipulate it, check if the user is authenticated or send back a response immediately. After this step we moved forward to work on the connection of the login page to the PostgreSQL database. We create a new file in the **/server/db** folder called *connect.js* which connects our log in page to the database server. We input the values of the port on which our database is running, username and password of the database along with the name of the database. Before doing this step, we created and set up our database separately on the PostgreSQL application. The database commands that were used for the table creation in PostgreSQL were also copy pasted inside a new file called "scripts.sql" in the main server folder.

To make sure the user credentials are matched to the credentials on the database, we created another file called "*common.js*" inside the **/server/utils** folder. This file makes use of JWT authentication. Jsonwebtoken or JWT, is the most used authentication protocol on the web. JWT is a web protocol that gets its name because it is based on the exchange of JSON files, which happen also to be called tokens, to provide authentication and authorization features. Now for the JWT authentication protocol to work seamlessly with the database we create a file called "*auth.js*" inside the **/server/middleware** folder. Middleware is a function that has access to request and response objects and extra next parameter. So, whenever we make an API request to the backend Express server, we can execute some code before or after the request is sent to the actual route. This file aids in verification of user credentials. Another "*auth.js*" file is created

after this step inside the **/server/routes** folder. This file contains the code which throws an error if the credentials are inconsistent or if they are incorrect. To query the PostgreSQL database, we have used a connection pool that handles multiple queries and gives a faster response. Inside /signup route inside *auth.js*, we first check if the data coming to the API contains the fields which are needed for registration using isInvalidField function defined in *common.js*. The data sent from the client will become available inside the req.body object and we get it as JSON because we have added app.use(express.json()) in *index.js*. Hence, all our current and future routes will receive data as JSON in req.body object. Inside the same file we have used We used the bcryptjs library for creating a secure password [to make sure the password is encrypted when it is being entered into the database]. The basic body of our login API is created here, and we have tested this out using POSTMAN. Postman is an application used for API testing. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain diverse types of responses that need to be subsequently validated.

The errors that are being thrown in the /server/routes/*auth.js* file are displayed on the web page through objects created inside the *errors.js* file. *Errors.js* file is first created inside the **/src/reducers** folder and then created inside the **src/actions** folder. This file is also connected to /src/actions/*auth.js* file along with *register.js* file in the backend; these files work together to display the error statements. After this step, our login API started working successfully. Meaning, when meant to throw out errors, it did. It was also successfully updating the database by taking advantage of the “CREATE" property of CRUD. The create functionality was working in the user registration part. Where if a new user was registered using the registration form, the new user’s credentials were added to the database.

Further Improvements:

To display a user’s profile page after logging in, a “*profile.js*” file was created inside the **/src/components** folder. This page lets the user change or update their first name and last name. To make sure this page works, another “*profile.js*” file was added inside the **/src/actions** folder to make sure it exports all the components are dispatched along with a third “*profile.js*” file inside the **/src/reducers** folder. After this step, the profileReducer component was added into the “*store.js*” file inside the **/src/store** folder. Routing to the profile page after logging in was made possible by adding a Profile component inside the src/router/*AppRouter.js* file. Now after the success of the profile page, we redirected the page using the “react-router-dom" library’s “history” method. The “history” package was imported inside *AppRouter.js* file along with the “react-router-dom”. To display the heading of the application I.e., B2C, we create a *header.js* file inside **/src/components** folder after which we add a header component inside our *AppRouter.js* file. The *common.js* file is then updated to make sure that the current session is maintained even

after hitting refresh by adding “const token = await generateAuthToken(user).” Inside this function, we have called the jwt.sign method by passing the userid and email for the first parameter. So, the generated jwt token contains these values to identify the user. To take out these values from the jwt token we have used the jwt-decode npm package which accepts the token and returns those values. Therefore, inside maintainSession we are using json_decode by passing a token and we are passing that extracted information to the updateStore function which dispatches the signIn action creator function. Furthermore, updates were made to make sure that the user does not go back to the login page or the registration page after signing in. To make sure that the profile is getting updated on the database too we created a profile API at the server-side to get and update profile information. To deploy this, we created a new file *profile.js* inside **server/routes** folder which contains the code to ensure that the database is getting updated each time the user changes their name. After this step, the route of this file was added into the “*index.js*” file inside the server folder.

To make sure that communication between the APIs is smooth and successful, we have used the axios library. Though this can also be achieved by other methods like fetch or AJAX, Axios can provide a little more functionality that goes a long way with applications that use React. Axios is a promise-based library used with Node. Axios was notably used in our src/utils/*common.js* file.

Now to implement a “LogOut” functionality, we made changes to the server/routes*/auth.js* file and actions/*auth.js* by adding a logout function. To make sure this functionality works, we add a file called “*Logout.js*” inside the **src/components** folder which is responsible for initiating logout feature. Logout route was also added to the *AppRouter.js* file.

By completing the above steps, we achieved a smooth API flow for our Login Page API. We then worked towards improving the login API to make sure that it is smoother and less buggy. Initially, we used the setAuthHeader() function call before making the API call for every private route like get or post/profile or get or post/login. It is a bit tedious to call the setAuthHeader() function every time before making an API call. Tedious because it may cause “AUTHENTICATION ERRORS”. We solved this problem by separating out the functionality that will do the job of calling that function and then proceed to make the API call. This functionality was separated by creating a “*api.js*” file inside the **/src/utils** folder. In the “*api.js*” file we send API URL as the first parameter, Data to be sent to the API as the second parameter, True as the third parameter if the authorization header needs to be set before making every API call and True as the fourth parameter if the authorization header needs to be removed after making every API call, to every GET, POST and PATCH route.

CRUD Functions

RESTful API has 4 basic functionalities for persistent storage: CREATE, READ, UPDATE and DELETE. When a resource has to be created, a HTTP POST request is sent to the API. The type of resource is determined by the request URL. Request body should contain a JSON object describing a resource to create. Then the initial state of the resource is determined by the body of the request when it is created. When some resources are created, they require certain properties to be provided whereas others can be created with empty JSON objects. In the READ function, resources are retrieved by performing HTTP GET requests. There are various methods to retrieve a resource but there are 2 main methods, one of them is requesting a list of resources, and the second method is when a single resource is requested. When we are requesting a single resource, it is marked by 'id' in the URL. But this behavior depends on the individual API endpoint. Now, the next functionality update is performed by using HTTP PATCH requests. These requests are issued to the URL where the resource is located at. When this request is performed, first properties of the request body are read, and only if the resource has a property with the same name, the property of the resource will be set to a new value. Deleting a resource can be done by sending a HTTP DELETE request to the URL where the resource is located at. This is the URL which contains the 'id' of the resource.

To build an API in Node.js, Express.js framework is needed. Express.js is fast, unopinionated, needs minimalist web framework for node.js, flexible. Express middleware can be used to address almost any task or the problem that we can think of. After creating a user called postgres and giving it a password, we created express.js for the server and node-postgres (pg) to connect to PostgreSQL. Now, the dependencies are loaded into node.modules and package.json.

As we have seen in the login structure and functionality part, *profile.js* component is created in components folder to display profile page once user login to the application. Another *profile.js* is created in the actions folder. In this *profile.js* we first imported BASE_API_URL, errors, history, update_profile and get and post. BASE_API_URL and UPDATE_PROFILE are imported from constants and get and post are imported from API.

Register Page:

We have imported a registered new user from the *auth.js* file. For registering a new user, it will ask a user to enter first name, last name, email, password, and confirm password as cpassword. If properties are not matching with any of the user's credentials in the database, it will display an error message. When registering any user, it will validate all the properties listed above: first name, last name, email id, password, cpassword. If any of this field is empty after pressing the register button, it will pop a message saying 'Please enter all the fields.' Or if the password and confirm password doesn't match it will pop up a message which says, 'password and cpassword doesn't match.' Once every field is correct, it will dispatch these properties and successMsg

would appear on the system which says, ‘User registered successfully.’ Now, since we have many different input fields like first name, last name, email, password, cpassword, we would normally do many different handleInputChange methods. All these handleInputChange do the same thing. They state a new input based on which input field is currently edited. Then data is extracted with the help of mapStateToProps.

Profile Page:

We have created a *profile.js* file inside **components** folder to display the profile page once a user login into the application. We have imported react, _, connect, form and button, from react, lodash, react-redux, and react-bootstrap respectively. Then we have imported initiateUpdateProfile from actions/profile to update the profile completely when it opens after logging in. To validate fields, validateFields is imported from utils/common. Profile will mount first name, last name, and email. If any previous properties are changed then this updated profile will change it, map it and the profile will be seen with all the updated information. It will check if all fields are entered, if not it will display an update error message which is ‘Please enter all the fields.’. Once all the data is entered, we will get a message ‘Profile updated successfully.’ on submitting all the data. handleOnChange is used to handle the user input in real time. It is used to track the value of input elements. handleOnChange is used here in *profile.js* to track the value of the first name, last name and email.

Then we have created *profile.js* in the actions folder to update profile data. After doing this, the update profile command is exported into utils*/constants.js. Profile.js* is also created in the reducers folder which is used to update a profile. We have created *profile.js* inside the reducers folder as well. In this *profile.js*, it will take the case update_profile as input and return updated profile state. ProfileReducer is added to store inside store*/store.js* so that one can access the profile information from the store in any component from store.profile using mapStateProps function. Once it is done, the profile component is also added to router/*AppRouter.js* file which gives the path for the profile component. Whenever one is clicking the login button, a login request is being sent to /signin API at the server and the data gets added from the response to the redux store. But once the login is successful, it is not directed to any other component.

API Table:

Operations	Resource URI	Method	Inputs	Returns
Create **Initiate Login** **Sign out**	*BASE_API_URL}/signup* *BASE_API_URL}/signin* *BASE_API_URL}/logout*	**POST**	Name, Email address, Password Email address, Password	**Profile** **Profile** **Login Page**
Read	*BASE_API_URL}/profile*	**GET**	All Account Details	**Updated profile**
Update	*BASE_API_URL}/profile*	**PUT**	Email address, Password, Name, Account Details	**Updated Profile**

Multiple operations are used in this project, but the 3 main basic operations are Create, Read, and Update. Each operation has its Resource URI, i.e. Resource Uniform Identifier, A Uniform Resource Identifier is a unique sequence of characters that identifies a logical or physical resource used by web technologies. Methods used are GET, POST, PUT. For using create operation, POST method is used along with its Resource Uniform Identifier which is BASE_API_URL}/signup. After going to this URI, Input- First Name, Last Name, Email id, Password, Cpassword is given, and the return is profile. If any of the fields is empty before the register button is hit, it will display a message-'Please enter all the fields.' For initiating login, BASE_API_URL}/signin is used. Inputs for initiating login would be email, password and then as a result it would display the profile. For signing out operations the result would be a login page. For retrieving the data, a read operation is used. Resource URI needed for this operation would be BASE_API_URL}/profile, it uses the GET method. It retrieves all account details such as First name, Last name, Email id, transaction history and in return, it gives an updated profile. BASE_API_URL}/profile is used for update operation as well. If one user has updated any of the fields mentioned above, it will get updated and the output will be updated profile. These 3 are the main basic functionalities used in this project to create, read, or update any resources that are needed.

Transaction Logic

Here we add the options to create a new bank account, allow users to deposit and withdraw amounts from the account, and allow users to generate and download statements in pdf format.

- We added *account* and *transactions* table scripts in **server/scripts.sql** which will be part of the Postgres database.

```
CREATE TABLE account(
    account_id BIGSERIAL PRIMARY KEY NOT NULL,
    account_no BIGINT NOT NULL,
    bank_name VARCHAR(50) NOT NULL,
    ifsc VARCHAR(32) NOT NULL,
    userid INTEGER NOT NULL,
    total_balance BIGINT NOT NULL DEFAULT 0,
    FOREIGN KEY(userid) REFERENCES bank_user(userid)
);
```

```
CREATE TABLE transactions(
    tr_id BIGSERIAL PRIMARY KEY NOT NULL,
    transaction_date TIMESTAMP NOT NULL,
    withdraw_amount DECIMAL NULL,
    deposit_amount DECIMAL NULL,
    balance DECIMAL NOT NULL DEFAULT 0,
    account_id BIGINT NOT NULL,
    FOREIGN KEY(account_id) REFERENCES account(account_id)
);
```

- We created a file called *Summary.js* inside the **components** folder in which we import libraries **React** from 'react' and **connect** from 'react-redux'. These libraries are part of React's new "hooks" APIs which give function components the ability to use local component states. Redux is an open-source JavaScript library for managing and centralizing the application state.

- We created a new file *AccountForm.js* inside the **components** folder. In this program, we declare a class AccountForm. React lets you define components as classes or functions. The only method you *must* define in a React.Component subclass is called render().

- We created a new file *Account.js* inside the **components** folder. We declared a React.component class called Account. Inside the class, we declare the state variables 'withdraw', 'deposit', and 'summary'. Here, we are displaying the buttons for the deposit, withdraw, and summary. Using the HTML tag which is then manipulated with the JavaScript code, we create three individual sections creating an onclick Event using Javascript's <button> tag. <Talk about variant tag>

  Example: `onClick={() => this.setSelectedType('deposit')}`
  In this, we have created a button which when clicked will redirect to the Deposit page.

- We add the route for /account inside the *AppRouter.js* file and also add the link for the Account component inside Header.js. We then added the forms to withdraw, deposit, and update account details in the *AccountForm.js* file. This was the tedious part because this involves the actual logic of our project. Inside the AccountForm class, we initialize the default state variables. We declared variables and using the **validateFields** function, we check the inputs given are valid.
  Example:

```
handleUpdateAccount = (ifsc) => {
  const fieldsToValidate = [{ ifsc }];

  const allFieldsEntered = validateFields(fieldsToValidate);
  if (!allFieldsEntered) {
    this.setState({
      errorMsg: {
        update_error: 'Please enter ifsc code.'
      }
    });
  } else {
    this.setState({
      errorMsg: ''
    });
  }
};
```

For variable 'ifsc' (similar to routing number), the validateFields function is responsible to check that the IFSC code is properly updated otherwise it should throw an error.

- We created a new file called *AddAccountForm.js* inside the **components** folder. In this, we created the option to add a bank account. After updating AddAccountForm.js, we checked if the account number exists and only then showed the form to withdraw or deposit otherwise, it automatically should navigate to the form for adding the account. To perform this function, we updated the code of *AccountForm.js*.

- We created a new file *account.js* inside the **src/actions** folder. The **export** statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the import statement. In this file, we create functions ***setAccount***, ***updateAccountBalance***, ***initiateGetAccntDetails***, ***initiateAddAccntDetails*** and ***initiateUpdateAccntDetails*** and we export the constant used for each function. Using try and catch, for each function, we check whether the requested data has been successfully dispatched otherwise it will give an error response.
- We created a new file *mask.js* inside the **src/utils** folder which contains the code to display only the last 4 digits of the account number and * will be displayed for other digits.

```
return number.slice(-4).padStart(number.length, '*');
```

- Create a new file *account.js* inside the **server/routes** folder. Note, we are exporting here, Router and getAccountByAccountId function so we can use it in another route file without the need of re-writing the same code. This is because we need to call ***getAccountByAccountId*** once while getting the transaction and another time while downloading the pdf report.

- Summary:
  When the user is on the "Add Account" form and updates all the details and clicks on the "Submit" button, we call the ***initiateGetAccntDetails*** function so that the user will be able to see the form to withdraw or deposit the amount. For that, we have a return statement inside the try block of *actions/account.js* for ***initiateAddAccntDetails*** function:

```
return await post(`${BASE_API_URL}/account`
```

- Since ***initiateAddAccntDetails*** is declared as an async function, the return keyword will return a promise from the function and in AccountForm.js, inside ***handleAddAccount*** function, we have added a .then handler to call the ***initiateGetAccntDetails*** function. This .then handler is responsible for the "Edit Account Details" link.

- We created a new file *transactions.js* inside the **server/routes** folder. This is where we actually do the withdraw and deposit function. Whenever we are depositing or withdrawing any amount, we are adding the details in the ***transactions*** table and also updating the total_balance column from the ***account*** table.
  Remember: If one of them fails then the other should not happen.
  If the transactions table is updated but the total_balance column value is not updated due to some error, then we need to revert the new data added in the transactions table and vice versa. So we are executing these two things as a transaction therefore instead of pool.query, we are using client.query here where the statements that need to be executed together will be added in between await client.query('begin') and await client.query('commit'). If any of the queries in between these calls fail, we call the await client. query('rollback') from inside the catch block so the other transaction will be reverted means either both will succeed or none of them. This will ensure that data will remain consistent in the table.

- We add the *getClient* function inside server/db/connect.js and export it from the file:

```
const getClient = async () => {
  try {
    const client = await pool.connect();
    return client;
  } catch (error) {
    return null;
  }
};
```

- We added transactions routes to the server/index.js file:

```
const transactionsRoute = require('./routes/transactions');
```

- We created a new file transactions.js inside the **src**/*actions* folder. Here we have created functions ***initiateDepositAmount*** and ***initiateWithdrawAmount***. Using try and catch, for each function, we check whether the ***updateAccountBalance*** function has been successfully dispatched otherwise it will give an error response.

- Now, you will be able to withdraw or deposit the amount to the bank account. We added a “logout” feature which is responsible to clear account information once the user logs out of the application.

Statement Generation

- After completing the withdraw and deposit account functionality, we talk about how to generate and download statements. In **components**/*Summary.js*, we create a React.component class called 'Summary'. We initialized the state variables *startDate (YYYY-MM-DD)*, *endDate (YYYY-MM-DD)*, *transactions* (this is a list that gets appended when you add the transactions between *startDate* and *endDate*), isDownloading (boolean flag), *formSubmitted* (boolean flag), and errorMsg.

- Open **src**/*index.js* and add 'react-bootstrap-table' and 'react-datepicker' CSS imports before main.scss import statement. React Bootstrap Tables are components with basic table features. They let you aggregate a huge amount of data and present it in a clear and orderly way. The React datepicker is a compelling and reusable component used for displaying the dates using a calendar dialog.

- We added ***setTransactions***, ***initiateGetTransactions***, and ***downloadReport*** functions inside actions/transactions.js file. These functions form the core of the statement generation algorithm.

- We created a new file Report.js inside the **components** folder in which we have written how to format the bootstrap- table to display the statements in table format.

- We created a new **views** folder inside the **server** folder and added *transactions.ejs* file. In this case, **ejs** is a templating engine we are using in Node.js to dynamically generate content.

- <u>Summary:</u>
  In the Statement Generation functionality, the user can select the date range for which you want the transaction report and click on the "Generate Report" button. If there are no transactions during that period, you will see a message - "No transactions found". If there are transactions within those periods, then you will see the transactions in table format and will be able to download the report as pdf.
- When we click the "Generate Report" button on the summary page, we are calling the /transactions/account_id route by passing the selected start date and end date and get the list of transactions and add it to the redux store and then we pass those transactions to *Report.js* component and display it in table format.

- Now, when we click on "Download Report" button, we are calling the /download/account_id route from routes/transactions.js file by where we are taking the transactions list and passing the transactions list to *transactions.ejs file* from server/views folder and using ejs.compile method to generate data with dynamic values from transactions array which is stored in the output variable and then we are writing that output html to the *transaction.html* file and then we call the generatePDF function to

generate the pdf file using puppeteer library by passing the generated transactions.html file. The generated transactions.html and transactions.pdf will be stored in the **views** folder.

- Then once the pdf is generated, to download the pdf from the server, we are using the ***downloadjs*** library by sending the content of pdf to download function provided by ***downloadjs*** library. The ***downloadjs*** library is very powerful, we can download any type of file by just passing the content of the file as the first parameter to the download function, file to name as the second parameter, and type of file as the third parameter.

Challenges

1. *AppRouter.js* : This library initially gave us an error along with the node-sass library because it was deprecated. We fixed it by downgrading the node.js version i.e. we downloaded the LTS version to fix this issue.
    - npm install react-router-dom@5
    - yarn add node-sass@4.14.1

2. Use of class and hooks: Initially we tried implementing the project using class components. We ran into multiple problems while trying to run the code. We found out that Class components are outdated and were replaced with hooks 2 years ago. So, we solved this problem by replacing the class components with the hooks.

3. .PHP connection error to postgres - shifted it to node: Initially we tried connecting the database to our application [front-end] using php. But we ran into a lot of problems due to lack of documentation with regards to connection of PostgreSQL to a javascript front end application. We researched and found out that php is not very compatible with postgres in general and it usually works well with mysql. We solved this problem by connecting the database to the front end using node.js and react.

4. Most of the problems regarding logical redirection were solved using POSTMAN. On numerous occasions we would run into navigation problems/ redirection problems while trying to navigate from one page to another. These problems were studied and solved by using POSTMAN and also a lot of discussions on stackoverflow.

5. Continuous debugging of library functions error, axios, authentication, react router dom, error 400,class errors, register variable errors. These errors were studied and fixed by going through discussions on stackoverflow and similar websites.

6. Making sure code is properly placed in the right directory/ environment variable errors: As node and react are directory platforms, it is very important to make sure the right code is in the right place. If there is even a slight mismatch in the location, then the application would fail to start. There were many instances where we wrote the code in the wrong directory. We fixed it by backtracking and going through our directories and files multiple times.

7. Database CRUD changes - Initially we tested our code using an ad-hoc database because our CRUD functions were not working properly. We rectified our code and wrote the

CRUD functionality in such a way that whenever you add a customer, it automatically updates to the database.

Future Work

1. Work on adding an analytics component to the application. This analytics widget would be designed to provide insights into customers' expenditures.

2. Make the application much more versatile by adding third party payment options like Google pay, Apple pay, etc. This would be implemented by using the API keys of the third party.

3. Make the application more secure by implementing better Identity and Access Management solutions.

4. Work more on the UX design of the application to make it more interactive.

5. Make the application cloud-based, for efficient performance and handling.

References

[1] https://en.wikipedia.org/wiki/React_(JavaScript_library)
[2] https://www.freecodecamp.org/news/is-react-a-library-or-a-framework/
[3] https://en.wikipedia.org/wiki/Node.js
[5]https://medium.com/@abesingh1/create-react-app-files-folders-structure-explained-df24770f852
[5] https://reactjs.org/docs/react-component.html
[6] https://www.w3schools.com/tags/tag_div.ASP
[7] https://www.w3schools.com/jsref/event_onclick.asp
[8] https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export
[9] https://www.sohamkamani.com/nodejs/sql-transactions/#basic-transactions
[10] https://www.thisdot.co/blog/connecting-to-postgresql-with-node-js
[11] https://mdbootstrap.com/docs/react/tables/basic/
[12] https://www.positronx.io/react-datepicker-tutorial-with-react-datepicker-examples/