VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Phat Tran

EXPENSE TRACKER APPLICATION USING MERN STACK

Technology and Communication 2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Phat Tran
Title	Expense Tracker Application Using MERN Stack
Year	2023
Language	English
Pages	56
Name of Supervisor	Kenneth Norrgård

This thesis demonstrates the design and implementation of an expense tracking web application. The product was created for the purpose of mastering full stack web development, as well as being a tool to keep track and adjust the author's spending habits.

The project was designed and developed using the MERN stack (MongoDB, ExpressJS, ReactJS, NodeJS). The front-end and the back-end were being coded simultaneously to ensure the proper connection between the integral parts of the entire website. The testing process was also being done constantly to prevent any error and bug that can affect the rest of the development process.

The result is a fully functional application with an easy-to-navigate user interface to input and track all the transactions, accompanied by a bar chart to give a more visualized perspective on the monthly expense. The final product also comes with an authorization system for security purposes.

Keywords[1] software engineering, application framework, expense, web development

TABLE OF CONTENTS

TABLE OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

MERN:	MongoDB, Express, React, Node.js
CRUD:	Create, Read, Update, Delete
API:	Application Programming Interface
NPM:	Node Package Manager
CSS:	Cascading Style Sheets
HTML:	Hypertext Markup Language
HTTP:	Hypertext Transfer Protocol
URL:	Uniform Resource Locator
JSON:	JavaScript Object Notation
JWT:	JSON Web Token
DOM:	Document Object Model
SPA:	Single Page Application
UI:	User Interface
GUI:	Graphical User Interface
CORS:	Cross-Origin Resource Sharing
ID:	Identification

1 INTRODUCTION

In a web developer's career, a full stack web application is an important asset in their portfolio. Besides demonstrating their proficiency in multiple programming languages, frameworks, and tools, it also showcases their knowledge about the structure and design philosophy of an application from the front-end to the back-end. For junior developers, their first full stack project does not have to be the most complex product. It can be a simple web application with CRUD (Create, Read, Update, Delete) methods and an authentication system. These elements are the foundation of any web-based platform, even the biggest names in the industry, such as Facebook, Twitter, Netflix. That is the reason why this project was chosen as the main topic of this thesis. By going through the thought process, conceptualizing, researching, coding, and debugging, the project has become an integral part of the author's study in web development.

2 BACKGROUND AND PURPOSE OF THE PROJECT

Struggling financially is a well-known issue among students. Due to the rise of energy cost, the price of daily necessities and food has also been increased tremendously. According to research from the National Union of Students (NUS) in July, in the United Kingdom, one in three students is left with less than £50 a month after paying rent and bills, and 96 per cent were cutting back on their spending as a result of the cost of living crisis (Staton, 2022). Many students ought to seek part-time jobs, in which the fields of work are not related to their fields of study, to support themselves financially. As a result, they need to balance out their work and their study to work effectively while maintaining acceptable grades at the universities. Eventually, budget management has become a crucial skill that every student must learn and practice during their years of study.

Budget management is a feature that can be seen in most of the banking applications, which allow users to view their monthly credit card usage in different categories, such as food, utility, entertainment, or travelling. From there, users can gain better insights into their spending habits and adjust accordingly to their budget. This is basically the whole concept of the application which is presented in this thesis.

The aim of this project is to build a mobile application utilizing the MERN stack (MongoDB, Express, React, and Node.js) that helps students in keeping track of their spending habits. The friendly user-interface of this application makes it easier for students to log their costs and track their spending trends over time. They may use this information to see where they might reduce their spending and improve their financial decisions.

The research problems that this project seeks to address include:

- How can the MERN stack be leveraged to create an efficient and user-friendly application for tracking student expenses?
- How can the MERN stack be used to create a scalable and secure application that can handle large volumes of financial data while maintaining user privacy?
- What features should the application include to make it user-friendly and efficient?

In this thesis, we will assess the functionality, usability, and security of the application and demonstrate how well it fulfills the requirements of users. The remaining parts of this thesis

are structured as follows: the third chapter provides the theoretical background of the all the technologies used during the development. The fourth chapter highlights the code that defines the core functionality of the application, including the design and architecture philosophy behind the written code. The fifth chapter showcases how the product works in its final stage. Finally, the sixth chapter concludes the thesis by summarizing key findings, learning outcome, and proposing future potential improvement that could be done on the project.

3 THEORETICAL BACKGROUND

3.1 Web Stack and the MERN Stack

A web stack is a collection of different programming languages and technologies which are combined to form a complete application. A typical web stack includes three layers:

- Client: As known as the front-end. This is the part which is visible to the user. From here, users can perform various interactions with the application which will be sent as requests to the server.
- Server: As known as the back-end. This part contains all the logic to handle the requests received and generate response back to the client.
- Database: This is where all the data (text, number, media, user credentials, etc.) is stored, organized, and retrieved from the server.

There are many web stacks in the industry, each has its own advantages and disadvantages. The choice of stack depends on the company's scale and the preferences of the developers.

One of the most widely used web stacks in recent years is the JavaScript based MERN stack. MERN stands for MongoDB, ExpressJS, ReactJS and NodeJS. ReactJS. We can see how the stack operates in Figure 1 below. ReactJS at the top layer acts as the front-end. MongoDB at the bottom layer is the data storage. While NodeJS, ExpressJS in the middle handles request and output the response to both the layers.

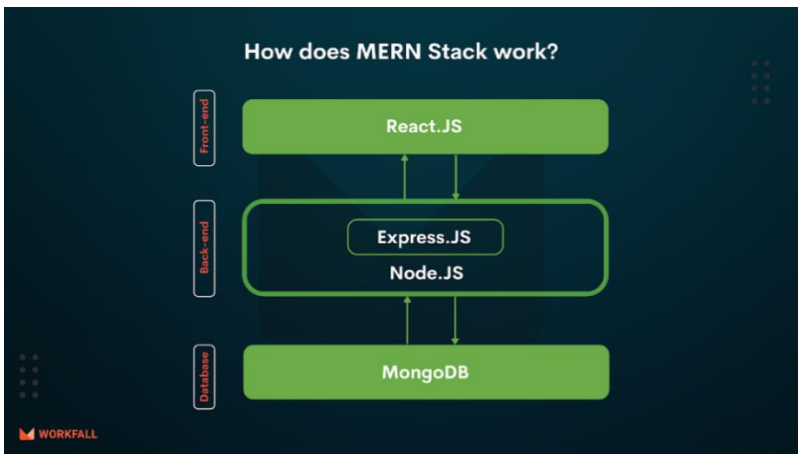


Figure 1 Diagram showing how the MERN stack works **(Workfall, 2022)**

3.2 Front-end

In web development, the front-end, as known as client, refers to the part of a web application where all the direct interactions happen between the user and the application. All the front-end technologies used in this project are described below.

3.2.1 ReactJS

ReactJS, simply known as React, is one of the most popular JavaScript libraries in the world. Introduced by Facebook (now known as Meta) in 2013, it has been widely used by web developers all over the world. It is an open-source project as a public repository on GitHub and is constantly maintained and updated by a community of developers. As we can see from the graph below, React is the second best choice of web framework in 2022, right behind Node.

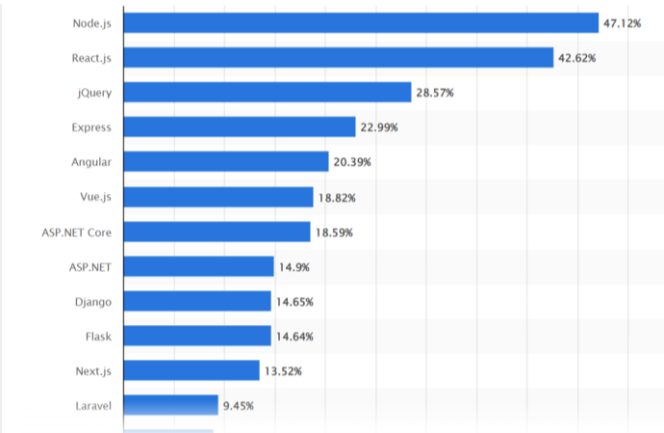


Figure 2 Most used web frameworks among developers worldwide, as of 2022 **(Vailshery, 2022)**

React utilized the concept of component-based design, which means that the user interface is divided into discrete, reusable modules known as components. Every component has a unique state that controls how it renders and functions. Moreover, each component can be passed in data, or "props," from its parent component.

Virtual Document Object Model (DOM) is one of the biggest advantages of React. The hierarchical structure of HTML components that make up a web page, known as the "real DOM," is replaced by the "virtual DOM," which is a simplified version of the real DOM. Rendering is accelerated and made more effective using React, which tracks UI changes using the virtual DOM and updates just the relevant portions of the real DOM.

From version 16.8 onwards, React has allowed developers to leverage hook, a function that can be used to manage state and the life cycle of functional components, which are components that are written in vanilla JavaScript functions. This makes the development process easier and the code more readable for other developers who work on the same project.

React also uses JavaScript Syntax Extension (JSX), an extension for JavaScript syntax that allows writing HTML code inside our JavaScript code, instead of having to use JavaScript functions to create and append HTML element to the DOM. This also helps reviewing code tremendously simpler because we can know exactly the rendering structure pf the component just by looking at the return function.

```
import React, { useState, useEffect } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);

  const incrementCount = () => {
    setCount(count + 1);
  };

  const decrementCount = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h1>Counter</h1>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>+</button>
      <button onClick={decrementCount}>-</button>
    </div>
  );
};

export default Counter;
```

Figure 3 Example of a React component with useState and useEffect hook

3.2.2 Material UI

Material UI is an open-source library with a user interface based on Google's Material Design system. It offers a selection of reusable and customizable UI elements that can easily be stylized using CSS syntax to fit the specific requirements of our React application. There is a wide variety of pre-built components in Material UI, including buttons, forms, navigation bars, dialog boxes, and more, which can speed up the development process significantly.

These components are well-tested and can be integrated seamlessly into our application, cutting down the requirement for stylizing using CSS, which can be troublesome and time-consuming.

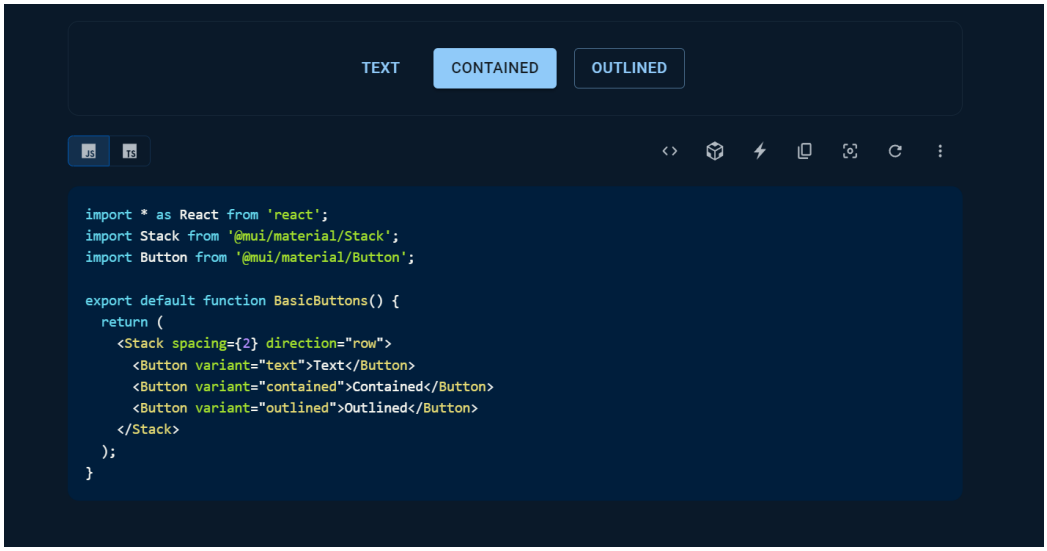


Figure 4 Example of a button component using Material UI **(Material UI, 2023)**

3.3 Back-end

The back-end, often known as server, is where all the logic that regards API, routing, authentication, and website infrastructure takes place off-screen. It plays a significant role as the bridge between the client and the database, by receiving inputs from the user and storing them to the database or retrieving the data from there to display it to the user.

3.3.1 NodeJS

NodeJS is an open-source, cross-platform, runtime environment built on the Chrome V8 JavaScript engine. It allows developers to run JavaScript code without the need of a web browser, with high compatibility to various operating systems like Windows, Linux or macOS.

Web servers, web apps, command-line tools, and other types of network applications can be created with NodeJS. Building scalable and real-time applications is suitable to NodeJS thanks to its event-driven, non-blocking input/output style.

NodeJS comes with Node Package Manager (npm), which grants users access to a massive ecosystem of modules and packages. According to Wikipedia, over 1.3 million packages are

available in the main NPM registry (Wikipedia, npm). The huge variety of choices for libraries and tools will serve any specific need of developers. One of the most famous among them is ExpressJS.

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, world!');
});

server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

Figure 5 Example of a simple web server built on NodeJS

3.3.2 ExpressJS

ExpressJS is a NodeJS web application framework which is lightweight, versatile, and offers a set of robust features for both web and mobile apps. By offering a thin layer of basic web application functionalities, such as routing, middleware, and HTTP handling, it is intended to make it easier than using vanilla NodeJS, to manage and arrange routes in complicated web applications.

ExpressJS is especially well-suited for the MERN stack because it provides developers the ability to quickly build RESTful APIs and handle HTTP requests from the client, such as the GET, POST, PATCH, DELETE method, which are the key features in web applications development. This is the reason why it takes the “E” in the MERN stack.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello, world!');
});

app.listen(port, () => {
  console.log(`Server listening on ${port}`);
});
```

Figure 6 Example of a simple web server built on ExpressJS

3.4 Database

Database is an organized collection of data stored and accessed electronically. Small databases can be stored on a file system, while large databases are hosted on computer clusters or cloud storage. A database is typically controlled by a database management system (DBMS), a software that interacts with end users, applications, and the database itself to capture and analyze the data. (Wikipedia, Database)

3.4.1 MongoDB

MongoDB is a document-oriented database, a type of NoSQL database, which is designed for scalability, high performance, and flexibility. MongoDB is a suitable choice for applications with fast changing or unpredictable data models since it stores data in JSON format documents, which can have easily customizable and dynamic schemas. Using MongoDB Atlas, users can access their data through a friendly GUI, without having to log in to their application to view the data. Besides auto-generating "_id" for each object, MongoDB also introduces a wide range of methods for querying and manipulating data, such as "find", "findById", "updateOne", "deleteOne". With flexible data model, simple query language, and strong community support, MongoDB is widely considered a good option for beginners who are new to database development.

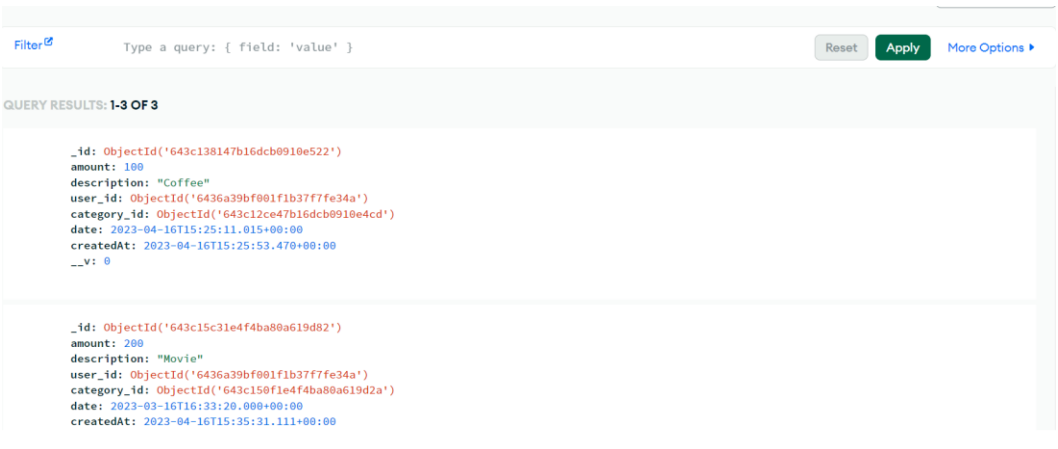


Figure 7 MongoDB Atlas user interface

3.4.2 Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and NodeJS. Mongoose provides a straight-forward, schema-based solution to model our application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box. (Mongoose)

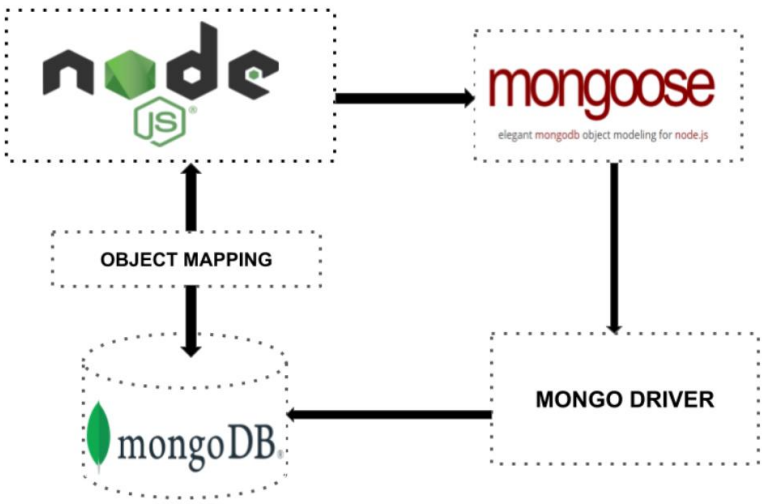


Figure 8 How Mongoose works in a server **(After Academy)**

3.5 Development Environment

3.5.1 Code Editor and Optional Extensions

When writing code, the most crucial tool is without a doubt the code editor since this is where we spend most of the time in during the development. The efficiency and productivity of the work may be improved, and a huge amount of time can be saved by using the right code editor in conjunction with useful extensions.

The choice of code editor for this project is Visual Studio Code (also known as VS Code), a light-weight source code editor created and published by Microsoft, with multiple platforms supports including Windows, Linux, macOS. VS Code provides various quality-of-life features such as syntax highlighting, code formatting, built-in terminal, code auto-completing, debugging, etc.

For this project, VS Code has been installed with these extensions, which furthermore enhanced the user experience. It should be noted that these extensions are just the author's preferences and are completely optional:

- Auto Rename Tag: automatically renames the closing tag to match the opening tag you just edited.
- CodeSnap: allows users to take a screenshot of their code by simply selecting the lines of code we want to capture then right click and choose "CodeSnap". This extension is significantly useful during the writing process of this thesis.
- Prettier: format tool for code's clean look and readability.
- Material Icon Theme: provide an intuitive and lively look for the files and folders in VS Code based on their names and extensions. This helps navigating through the explorer tab easier and faster.

3.5.2 Version Control

Version control is a system that enables many developers to work collaboratively on a project by tracking changes made to the project and offering tools for combining those changes. These tracking mechanisms are made possible by a record of the entire codebase that is generated each time a change is made. This record is then saved as a new version of the codebase, along with metadata which describes the changes that were made, who made them, and when they were made. With these records, developers can easily roll back the project to a previous version if the code does not work as before and they are unable to figure out which change they made is responsible for that.

In this project, we will use Git, one of the most well-known version controls which is easy to set up and configure. In combination with Git, we will use GitHub, a web-based platform that provides a hosting service for our project. We will create a repository on the platform. Then, after a remarkable change has been made to our product, for example a new feature added, we will commit our code, which will create a record of the current state of our entire codebase. Then we will push the commit to GitHub, synchronizing the repository on our local storage with the one on GitHub's cloud storage. The platform also has a user-friendly UI that allows users to review all the changes made to the code including which line was deleted, which was added.

3.5.3 Installed Dependencies

Below is the list of all the crucial Node packages and frameworks that were installed for this project. Except for Node and React, all the dependencies can be installed with the command "npm install <name of dependency>".

- Node (version 16.15.0): downloaded the installed from Node's official website. Command "npm init" was used to initialize the back-end.
- React (version 18.2.0): initialized with command "npx create-react-app client" to create the base code for the front-end.
- @devexpress/dx-react-chart-material-ui: required for creating the column chart from the data.
- @emotion/react: required for inline-styling for Material-UI components.
- @emotion/styled: required for Material-UI's Styled Component.
- @mui/icons-material: required for Material-UI's icon package.
- @mui/material: main package of components from Material-UI.
- @mui/x-date-pickers: required for Material-UI's date selection component.
- dayjs: required for day value formatting.
- js-cookie: allows us to access the cookie from the browser.
- redux: allows us to store and access the state of an object from any component in the application, without having to rely on passing value in props through multiple components.
- @reduxjs/toolkit: simplifies the process of creating Redux's store and its reducer function, action.
- react-router-dom: required to build the routing feature, which allows us navigating between different pages or views in our React application.
- bcrypt: used for password hashing, ensuring the security of the application.
- body-parser: used for extracting the requests body data from incoming POST or PATCH requests from the client.
- cors: stands for Cross-Origin Resource Sharing, a security mechanism that is embedded in the web browsers to prevent a web page from making requests to a different domain. Installing "cors" allows all origins to create requests to our server.

- dotenv: allows us to store confidential environment variables such as: URL, username, password, to a “process.env” object in a “.env” file.
- express: responsible for all the routing and controllers of our APIs.
- jsonwebtoken: allow us to generate and verify JSON Web Tokens (JWT) for authorization and authentication.
- mongoose: Object Data Modeling (ODM) library for MongoDB and NodeJS.
- nodemon: automatically restarts the server when we make changes and save our code.
- passport: provides a simple and flexible authentication middleware for web applications.
- passport-jwt: is a strategy (a method for handling a specific type of authentication) for “passport” that allows you to authenticate users using JWTs.

4 APPROACH AND IMPLEMENTATION

The application was finished in one month and a half, which includes the time for designing, planning, reading the documents, coding, debugging and deployment. The design and planning phase determined how the final product would look like, what pages are and what components will be used on a page. The front-end and the back-end were being developed simultaneously to make sure they are connected appropriately to each other. Various threads and posts on forums such as Stack Overflow and Reddit played a huge role in the development of the application, providing numerous contexts and solutions to the problems encountered. The writing process for this thesis started as soon as the product was finished.

4.1 Application Workflow

Figure 16 shows the workflow of our application. Users start at the login page. If an account has already been created, they can log in to proceed to the home page. If not, they are required to create an account on the register page. After signing up, they will be redirected to the login page to enter the user credentials they just created. Landing at the home page after successfully logged in, they can navigate back and forth between there and the categories. From both pages, users can log out which sends them back to the login page. Both the home and category page are “protected routes” which can only be accessed if the user is logged in. While the sign in and register page can be visited by anyone, which is why they are referred to as “guest routes”.

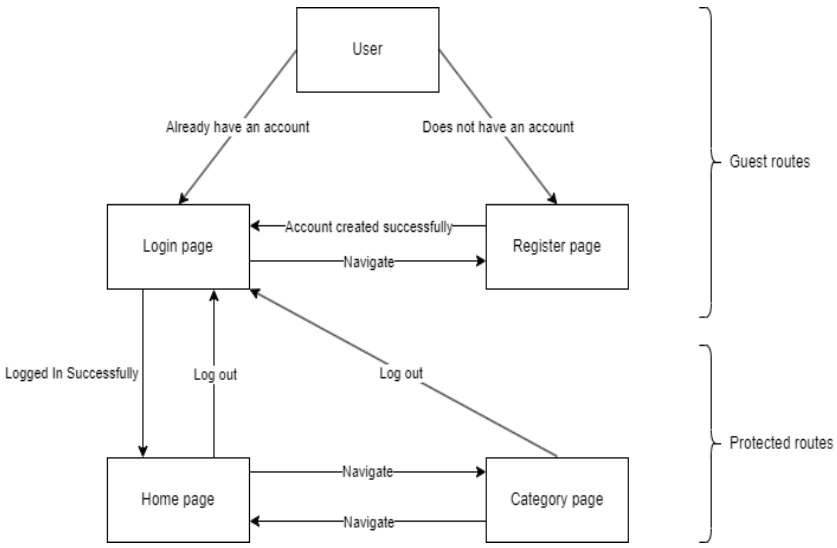


Figure 9 Workflow of the application

Despite navigating through multiple pages, in reality, users only stay on one page the entire time. This is a well-known trait from JavaScript frameworks like React, called single page application (SPA). As the users open the application, only one HTML page is loaded and used throughout the whole session. React dynamically makes change to the content on the page by updating the virtual DOM as the users interact with the application, without reloading or rendering a new page. As a result, this provides a faster and more seamless experience for the users, while saving a lot of resources required for the server to operate.

4.2 Project Structure

From the screenshot of the project tree below, we can have a clear overview of our project structure. Inside of our top folder is two main subdirectories: “client”, which obviously contains the front-end logic and “server”, where all the back-end code resides. At the same level of the two subdirectories is a “.gitignore” file, which signals Git to not include “node_modules”, a large in size folder that contains all the dependencies, and the “.env” files from both client and server.

In the “client” folder, there are two subfolders: “public” and “src”. The “public” folder contains all the static assets such as images, fonts and other files which are served directly to the browser. The folder was created by “create-react-app” command and can be viewed publicly by anyone who has the URL to our application. The ”src” directory, which is a short way to say “source”, holds several notable children folders:

- The “components” folder contains all the modular and reusable components that are used on a page, such as forms, lists, charts, and the navigation bar.
- The “pages” folder is where all the pages in the navigation system can be seen, including the login, register, home, and category page.
- The “store” folder is where we configure our Redux store and its reducers.
- The “utils” folder holds our redirection components which redirect user to destinations based on their authorization status.

While in the “server” folder, there are also multiple subdirectories that worth mentioning:

- The “config” folder contains all the configurations for our PassportJS middleware for authorization.

- The “controller” folder includes all the functions in the server that tie to our routes.
- The “database” folder is where we connect to our MongoDB database.
- The “model” folder holds all the schemas that describe the structure and content of our data in MongoDB.
- The “routes” folder is where we define all the APIs with its routing. Each route ties to an HTTP method with a function from our controllers.

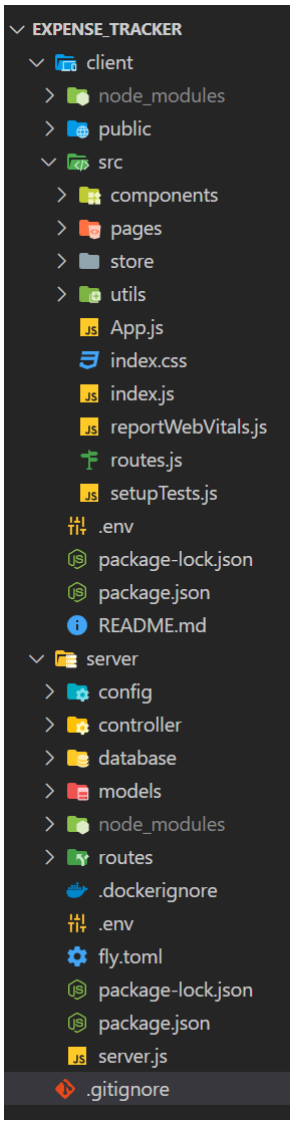


Figure 10 The project tree of the application

4.3 Back-end Logic

The figure below shows all the content in the back-end, in which there are several crucial files and folders that are worth mentioning.

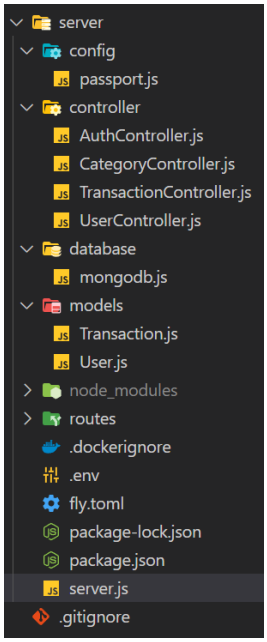


Figure 11 The server folder with its subfolders and files

The entry point of our server is the “server.js” file. In this file, after retrieving all the values from the “.dotenv” file, defining a “PORT” value and initializing an “app” constant with Express, we apply multiple middlewares to the application including “cors”, “bodyparser” and “passport”. The “passport” middleware uses a configuration in “passport.js” from the “config” folder. After that, we define our first route “/” which leads us to many sub-routes that were packaged in a routes value. Finally, we use the imported “connect” function to start our server and print the status message to our console. There is a lot to be unpacked here and we will go through each value that was imported from outside this file.

```
import express from "express";
import cors from "cors";
import bodyParser from "body-parser";
import connect from "./database/mongodb.js";
import passport from "passport";
import passportConfig from "./config/passport.js";
import * as dotenv from "dotenv";
import routes from "./routes/index.js";

dotenv.config();

const PORT = process.env.PORT || 4000;
const app = express();

app.use(cors());
app.use(bodyParser.json());
app.use(passport.initialize());
passportConfig(passport);

app.use("/", routes);

await connect();

app.listen(PORT, () => {
  console.log("Server is running at http://localhost:4000");
});
```

Figure 12 The "server.js" file

4.3.1 Authentication with PassportJS

The “passport.js” file is a module that is used to authenticate user requests with JWTs. By using the “passport-jwt” package, it defines a strategy called “JwtStrategy” for Passport to use. The strategy is called each time a request from the client is made to a protected route. It then extracted the JWT from the ‘Authorization’ header of the request, as a Bearer token with a secret key used to sign the JWT, which is obtained from the “.env” file. If the JWT is valid and the user can be found in the database (by using the imported “User” model), the “done” function is called with the authenticated user object. Otherwise, it will be called with a false value to indicate that authentication failed, preventing users from proceeding to that route.

```
import pkg from "passport-jwt";
const JwtStrategy = pkg.Strategy;
const ExtractJwt = pkg.ExtractJwt;
import User from "../models/User.js";
import * as dotenv from "dotenv";

dotenv.config();

let opts = {};

opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();
opts.secretOrKey = process.env.JWT_SECRET;

export default (passport) => {
  passport.use(
    new JwtStrategy(opts, function (jwt_payload, done) {
      User.findById(jwt_payload._id, function (err, user) {
        if (err) {
          return done(err, false);
        }
        if (user) {
          return done(null, user);
        } else {
          return done(null, false);
        }
      });
    })
  );
};
```

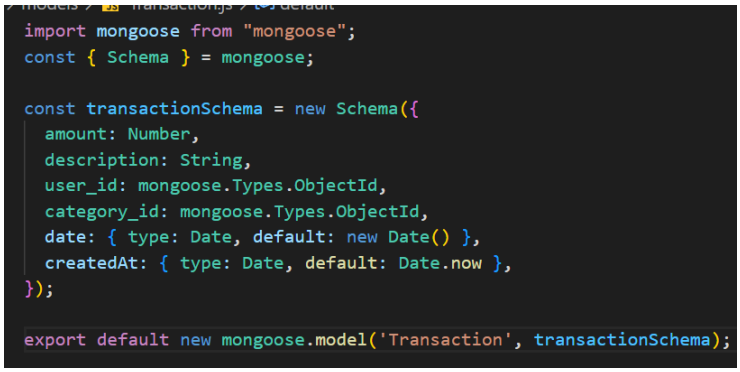
Figure 13 The "passport.js" file

4.3.2 MongoDB’s Data Modeling

A “Transaction” schema was created using Mongoose to define the transaction object with its own properties and export it as a data model in the database. The schema defines the following fields:

- “amount”: a number representing the transaction amount.
- “description”: a string describing the transaction.

- “user_id”: an “ObjectId” generated by “mongoose” to reference the user who made the transaction.
- “category_id”: an “ObjectId” generated by “mongoose” to reference the transaction category.
- “date”: a Date object stating the date of the transaction was made.
- createdAt: a Date object stating the date the transaction was added to the database, which is the current date.



```
import mongoose from "mongoose";
const { Schema } = mongoose;

const transactionSchema = new Schema({
  amount: Number,
  description: String,
  user_id: mongoose.Types.ObjectId,
  category_id: mongoose.Types.ObjectId,
  date: { type: Date, default: new Date() },
  createdAt: { type: Date, default: Date.now },
});

export default new mongoose.model('Transaction', transactionSchema);
```

Figure 14 The "Transaction" schema

On the other hand, the “User” schema defines the following fields:

- “firstName”: a string representing the user's first name.
- “lastName”: a string representing the user's last name.
- “email”: a string representing the user's email address.
- “password”: a string representing the user's password.
- “categories”: an array of objects containing two fields, “label” and “icon”, representing the user's transaction categories.

All the fields which have the property “required” must be filled in, otherwise it will display an error message in the console which is stated in the code. The schema also has a second argument which is an object with a timestamps field set to “true”. By using this object, Mongoose automatically adds two fields to each user data model:

- “createdAt”: a Date object representing the creation date of the user document.
- “updatedAt”: a Date object representing the last modification date of the user document.

```
import mongoose from "mongoose";
const { Schema } = mongoose;

const userSchema = new Schema(
  {
    firstName: { type: String, required: ["First name required"] },
    lastName: { type: String, required: ["Last name required"] },
    email: { type: String, required: ["Email required"] },
    password: { type: String, required: ["Password required"] },
    categories: [{ label: String, icon: String }],
  },
  { timestamps: true }
);

export default new mongoose.model("User", userSchema);
```

Figure 15 The "User" schema

4.3.3 Server's Controllers, Routing, and APIs

The "AuthController.js" file, which handles the authentication process for users, has two main functions: "login" and "register". Both are exported as parts of the modules of the server.

The register function is an asynchronous function that has two arguments, "req", which is the request sent from the client; and "res", which is what the server will response back. We will see this pattern applied to all the controller functions in this project. The "register" function checks if a user with the given email already exists in the database by calling the "findOne" method on the import "User" data model. If the user exists, it returns a response with an HTTP status code of 406 and a JSON object with the message "User already existed". If the user does not exist, the function hashes the provided password using the "bcrypt" library's "genSaltSync" and "hashSync" methods. Then it creates a new User object and sets its properties with the provided "email", "password", "firstName", "lastName", and "categories". Next, it saves the user data in the database by applying the "save" method on "User". Finally, it returns a response with an HTTP status code of 201 with the message "User created".

The login function uses the method "findOne" on "User" to check if a user with the given email already exists and will send status code 406 with the message "Credentials not found", if a user cannot be found in the database. If the user exists, the function compares the provided password with the stored hashed password using the "compare" method from the "bcrypt" library. If the passwords do not match, it returns status code 406 with the message "Wrong password". If the passwords match, the function creates a "payload" object with the user's email and id, signs the payload with a JWT using the "jsonwebtoken" library's "sign"

method, and returns status code 200 with the message "Successfully logged in", the signed token, and the user object.

```
import User from "../models/User.js";
import jwt from "jsonwebtoken";
import bcrypt from "bcrypt";

const categories = [
  { label: "Transportation", icon: "🚗" },
  { label: "Shopping", icon: "🛒" },
  { label: "Bills", icon: "🧾" },
  { label: "Investment", icon: "📈" },
];

export const register = async (req, res) => {
  const { email, password, firstName, lastName } = req.body;
  const userExists = await User.findOne({ email });
  if (userExists) {
    res.status(406).json({ message: "User already existed" });
    return;
  }

  const saltRounds = 10;
  const salt = bcrypt.genSaltSync(saltRounds);
  const hashedPassword = bcrypt.hashSync(password, salt);
  console.log(hashedPassword);

  const user = await User({
    email,
    password: hashedPassword,
    firstName,
    lastName,
    categories,
  });
  const savedUser = await user.save();
  console.log(savedUser);

  res.status(201).json({ message: "User created" });
};

export const login = async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email });
  if (!user) {
    res.status(406).json({ message: "Credentials not found" });
    return;
  }

  const matched = await bcrypt.compare(password, user.password);
  if (!matched) {
    res.status(406).json({ message: "Wrong password" });
    return;
  }

  const payload = {
    username: email,
    _id: user._id,
  };
  const token = jwt.sign(payload, process.env.JWT_SECRET);
  res.json({ message: "Successfully logged in", token, user });
};
```

Figure 16 The "AuthController.js" file

The “CategoryController.js” file defines three functions: “destroy”, “create”, and “update”, which alters the categories property of the user object.

The “destroy” function retrieves the categories array from the authenticated user, then filters out the category with the “_id” value that matches the “id” parameter in the request. After that, it uses the “updateOne” method on “User” model to update the user's categories array in the database. Finally, it returns a response with the updated user as a JSON object.

The “create” function retrieves the “label” and “icon” values from the request body. It then updates the user’s categories array by appending a new object with “label” and “icon” as its properties, using the “updateOne” method. Finally, it returns a JSON object containing the updated response object.

The “update” function also retrieves the “label” and “icon” values from the request’s body and updates the user’s categories array. But instead of appending new objects to the array, it maps through the array and updates the object that matches the “id” parameters in the request with the new “label” and “icon” values. Then, after applying the “updateOne” method on “User” model, it also returns the updated response object in a JSON object.

```
import User from "../models/User.js";

export const destroy = async (req, res) => {
  const categories = req.user.categories;
  const newCategories = categories.filter(
    (category) => category._id != req.params.id
  );
  const user = await User.updateOne(
    { _id: req.user.id },
    { $set: { categories: newCategories } }
  );
  res.json({ user });
};

export const create = async (req, res) => {
  const { label, icon } = req.body;
  const response = await User.updateOne(
    { _id: req.user.id },
    { $set: { categories: [...req.user.categories, { label, icon }] } }
  );
  res.json({ response });
};

export const update = async (req, res) => {
  const { label, icon } = req.body;
  const response = await User.updateOne(
    { _id: req.user.id },
    {
      $set: {
        categories: req.user.categories.map((category) => {
          if (category._id == req.params.id) {
            return { label, icon };
          }
          return category;
        }),
      },
    }
  );

  res.json({ response });
};
```

Figure 17 The "CategoryController.js" file

The “TransactionController.js” file is a representation of a full set of CRUD operations that user can perform on the “Transaction” data model.

The “index” function retrieves all transactions for the logged-in user and groups them by month using the MongoDB’s “aggregate” method. It first filters the transactions that belong to the logged-in user by matching the “user_id” field with the “$match” operator. Then, it groups the transactions by month using the “$group” operator, which creates a new field

“_id” that represents the month extracted from the date field using the “$month” operator. It also includes an “transactions” array that states the details of each transaction, as well as a new field “totalExpenses” that represents the sum of all transaction amounts for that month. Finally, it sorts the results by the “ _id” field in ascending order.

The “create” function creates a new transaction by extracting the input fields from the request body and saving a new instance of the “Transaction” model, with the “save” method. It adds a “user_id” field to the transaction object, which is set to the logged-in user's ID.

The “destroy” function handles deleting a transaction by its “_id” field, by applying the “deleteOne” method on “User”.

The “update” function updates a transaction in the database also by its “_id” field. It uses the “$set” operator to update the fields received from the request body.

```
import Transaction from "../models/Transaction.js";

export const index = async (req, res) => {
  const transaction = await Transaction.aggregate([
    {
      $match: { user_id: req.user._id },
    },
    {
      $group: {
        _id: { $month: "$date" },
        transactions: {
          $push: {
            amount: "$amount",
            description: "$description",
            date: "$date",
            _id: "$_id",
            category_id: "$category_id",
          },
        },
        totalExpenses: { $sum: "$amount" },
      },
    },
    { $sort: { _id: 1 } },
  ]);
  res.json({ data: transaction });
};

export const create = async (req, res) => {
  const { amount, description, date, category_id } = req.body;
  const transaction = new Transaction({
    amount,
    description,
    user_id: req.user._id,
    date,
    category_id,
  });
  await transaction.save();
  res.json({ message: "Success" });
};

export const destroy = async (req, res) => {
  await Transaction.deleteOne({ _id: req.params.id });
  res.json({ message: "Success" });
};

export const update = async (req, res) => {
  await Transaction.updateOne({ _id: req.params.id }, { $set: req.body });
  res.json({ message: "Success" });
};
```

Figure 18 The "TransactionController.js" file

The ”UserController.js” file is rather simple with only one function ”index” that responses a user object in JSON format, which is used to verify the user’s authorization.

```
export const index = (req, res) => {
  res.json({ user: req.user });
};
```

Figure 19 The "UserController.js" file

All the data model controllers are imported to be used in each of its corresponding API. The ”Auth Controller.js” goes to the ”AuthApi.js” file. The ”CategoryController.js” goes to the ”CategoryApi.js” file. While the ”TransactionController.js” goes to the ”TransactionApi.js” file. Finally, the ”UserController.js” goes to the ”UseApi.js” file. In each API, a ”Router” function provided by the ”express” library is initialized and assigned to a ”router” value. Then, each of the HTTP methods of ”router” will take two parameters, an endpoint and a function from the controller file. For example, in Figure 20 below, the ”post” method of the transaction API has an endpoint at ”/” and the ”create” function from the transaction controller attached to the method. In cases where an ID is required to send the request, an ”id” parameter is included in the endpoint. Then, after all the routes in the API have been set up, the API is exported as a module that will be accessed later.

```
import { Router } from "express";
import * as TransactionController from "../controller/TransactionController.js";

const router = Router();

router.get("/", TransactionController.index);
router.post("/", TransactionController.create);
router.delete("/:id", TransactionController.destroy);
router.patch("/:id", TransactionController.update);

export default router;
```

Figure 20 The "TransactionApi.js" file

```
import { Router } from "express";
import * as CategoryController from "../controller/CategoryController.js";

const router = Router();

router.delete("/:id", CategoryController.destroy);
router.post("/", CategoryController.create);
router.patch("/:id", CategoryController.update);


export default router;
```

Figure 21 The "CategoryApi.js" file

```
import { Router } from "express";
import { register, login } from "../controller/AuthController.js";

const router = Router();

router.post("/register", register);
router.post("/login", login);

export default router;
```

Figure 22 The "AuthApi.js" file

In one special case with the with the ”UserApi.js” file, the ”get” method also takes an additional parameter which utilizes the Passport’s ”authenticate” method with the ”jwt” strategy to check if the user is authenticated. The option ”session” is set to ”false”, which means that the server does not store user sessions on the server after they are authenticated, and instead will verify the JWT attached with every request.

```
import { Router } from "express";
import passport from "passport";
import * as UserController from "../controller/UserController.js";

const router = Router();

router.get(
  "/",
  passport.authenticate("jwt", { session: false }),
  UserController.index
);

export default router;
```

Figure 23 The "UserApi.js" file

After all the APIs with their routes have been defined. They are imported to the ”index.js” file, which acts as a hub to hold all the modules in one place. As all the APIs, a ”Router” function is also initialized and assigned to a ”router” value. The ”router” will use the method ”use” from the ”express” library, which takes two parameters, and endpoint and its corresponding API. The Passport’s ”authenticated” method mentioned above is assigned to a ”auth” value, which is passed as an additional parameter in the transaction and category route for authorization. After all the routes to the APIs have been set, the file is exported as a module to be used in the ”server.js” file.

```
import { Router } from "express";
import passport from "passport";
import TransactionsApi from "./TransactionsApi.js";
import AuthApi from "./AuthApi.js";
import UserApi from "./UserApi.js";
import CategoryApi from "./CategoryApi.js";

const router = Router();

const auth = passport.authenticate("jwt", { session: false });

router.use("/transaction", auth, TransactionsApi);
router.use("/auth", AuthApi);
router.use("/user", UserApi);
router.use("/category", auth, CategoryApi);

export default router;
```

Figure 24 The "index.js" file

Below is the list of all the routes with their HTTP methods that were used in this project:

HTTP Methods	End Point	Description	Access
GET	/transaction/	Fetch all transactions	Private
POST	/transaction/	Create a new transaction	Private
DELETE	/transaction/:id	Delete a transaction	Private
PATCH	/transaction/:id	Edit a transaction	Private
DELETE	/category/:id	Delete a category	Private
POST	/category/	Create a category	Private
PATCH	/category/:id	Edit a category	Private
POST	/login/	Log in with an account	Public
POST	/register/	Create a new account	Public
GET	/user/	Get information of the current user	Private

Figure 25 All routes and their HTTP methods in the application

4.3.4 Connecting to the Database

Ultimately, the ”mongdodb.js” file establishes connection to the MongoDB database. Each value, ”username”, ”password”, and ”url” is extracted from our ”.env” file and inserted into the URL linked to the database. Mongoose uses the method ”connect” to access the URL, then informs the status of the connection.

```
import mongoose from "mongoose";

async function connect() {
  const username = process.env.MONGO_DB_USERNAME;
  const password = process.env.MONGO_DB_PASSWORD;
  const url = process.env.MONGO_DB_URL;
  await mongoose
    .connect(
      `mongodb+srv://${username}:${password}@${url}/?retryWrites=true&w=majority`
    )
    .then(() => console.log("MongoDB connection is successful"));
}

export default connect;
```

Figure 26 The "mongodb.js" file

4.4 Front-end Logic

Figure 27 below shows all the files and folder in the front-end:

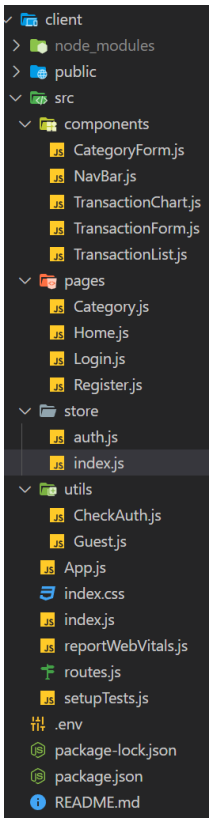


Figure 27 The client folder with its subfolders and files

In the ”client” directory, the ”index.js” file is the entry point of the front-end. This file was created by the default with the ”create-react-app” command, with some adjustment made to fulfill the specific needs of our application. It creates a root element using the ”createRoot” method of ReactDOM (a package that binds React with the DOM), and passes in an HTML element with an id of ”root”. Then, by default, the ”root” element would render the ”App” component inside a ”React.Strictmode” component. But in the case of our application, it renders a ”Provider” component from ”react-redux” library, that passes the imported ”store” object as a prop, in which we store our user’s state. The ”Provider” wraps around our ”RouterProvider” component which passes the imported ”router” object as a prop, which defines the routing functionality of our front-end.

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import { RouterProvider } from "react-router-dom";
import { Provider } from "react-redux";
import reportWebVitals from "./reportWebVitals";
import router from "./routes";
import store from "./store/index.js";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
    <Provider store={store}>
      <RouterProvider router={router} />
    </Provider>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Figure 28 The "index.js" file

4.4.1 State Management with Redux Toolkit

In our application, we store our user’s state using Redux with the help of Redux Toolkit, which makes configuring Redux easier with less boilerplate code and syntax. With Redux, we can avoid having to fetch the current user every time we need to access the state of the user from any component. Using Redux also eliminates prop drilling, the process of passing data from one component via several interconnected components to the component that needs it. Prop drilling results in long and unclean code. Consequently, there are greater possibilities for mistakes such as renaming the props midway by mistake, refactoring some data's structure, props being forwarded more often than is necessary, using default props unfairly or using default props unfairly or insufficiently (Scaler, 2023). Figure 29 below shows how the data is

passed in the prop drilling process, while Figure 30 demonstrates how data is access with Redux.

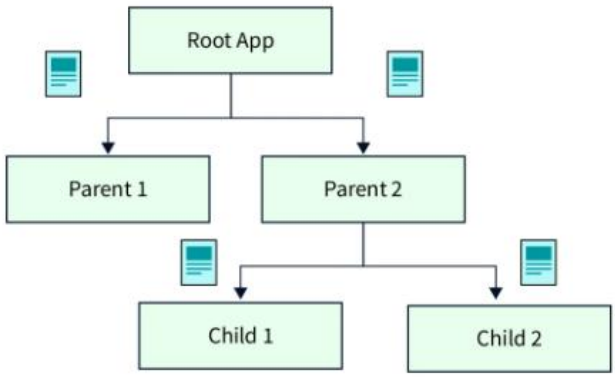


Figure 29 How prop drilling works

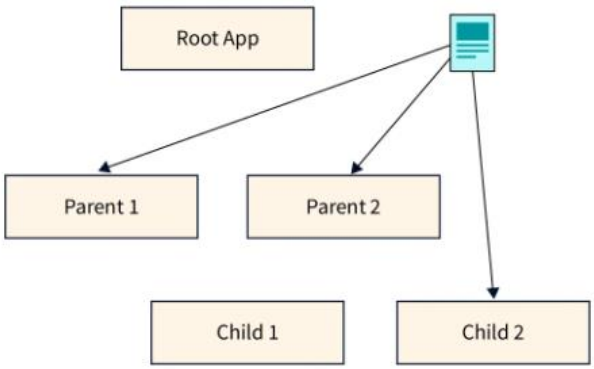


Figure 30 How Redux works

The ”index.js” file in the ”store” folder is where we set up our Redux store. We use the ”configureStore” function provided by Redux Toolkit, which takes an object specifying the ”reducer” property. ”Reducer” is a built-in function that takes the current state and an action and returns a new state, in which the imported ”authReducer” function is passed as a value to the ”auth” key. The ”authReducer” function handles any dispatched actions to the store and updates the ”auth” state in the store.

```
import { configureStore } from "@reduxjs/toolkit";
import authReducer from "./auth.js";

export default configureStore({
  reducer: {
    auth: authReducer,
  },
});
```

Figure 31 The "index.js" file in the "store" folder

In the ”auth.js” file, we defined and exported an ”authSlice ” object, which was created using the ”createSlice” function from Redux Toolkit. With ”createSlice”, developers can define a slice of the Redux store in a more compact way, by automatically generating action creators and reducers based on the provided state and reducer functions. The ”authSlice” object has three properties:

- ”name” is a string which is used to call the generated action types.
- ”intialState” is an object which has two properties set to their default values: ”isAuthenticated” which is initially set to ”false” and ”user” is an empty object.
- ”reducers” is an object that has a collection of actions which can be dispatched to alter the slice’s state. For our application, we defines two actions: ”setUser” which set ”user” property of the slice to the ”user” property from the payload and set ”isAuthenticated” to ”true”; ”logout” which empties the ”user” property and set ”isAuthenticated” to ”false”.

Both ”setUser” and ”logOut” are exported as the ”authSlice.actions” object. Finally the whole slice is exported as a generated ”reducer” function.

```
import { createSlice } from "@reduxjs/toolkit";

export const authSlice = createSlice({
  name: "auth",
  initialState: {
    isAuthenticated: false,
    user: {},
  },
  reducers: {
    setUser: (state, actions) => {
      state.user = actions.payload.user;
      state.isAuthenticated = true;
    },
    logOut: (state) => {
      state.user = {};
      state.isAuthenticated = false;
    },
  },
});

export const { setUser, logOut } = authSlice.actions;
export default authSlice.reducer;
```

Figure 32 The "auth.js" file

4.4.2 Routing in the Front-end

Figure 33 shows the code in the ”routes.js” file, which gives us an overview of the routing functionality of our application in the front-end, which was created by the ”createBrowserRouter” by React Router. All the pages and conditional rendering components were imported to serve as specific routes. ”App” is the top component which takes all the

remaining pages as its children. Both the ”Login” and ”Register” pages, which have ”/login” and ”/register” as their endpoints, respectively, are wrapped inside the ”Guest” components. While ”Home” and ”Category”, which take ”/” and ”/category”, are wrapped inside the ”CheckAuth” components.

```
import { createBrowserRouter } from "react-router-dom";
import App from "./App";
import Login from "./pages/Login";
import Register from "./pages/Register";
import Home from "./pages/Home";
import Category from "./pages/Category";
import CheckAuth from "./utils/CheckAuth";
import Guest from "./utils/Guest";

export default createBrowserRouter([
  {
    element: <App />,
    children: [
      {
        path: "/",
        element: (
          <CheckAuth>
            <Home />
          </CheckAuth>
        ),
      },
      {
        path: "/login",
        element: (
          <Guest>
            <Login />
          </Guest>
        ),
      },
      {
        path: "/register",
        element: (
          <Guest>
            <Register />
          </Guest>
        ),
      },
      {
        path: "/category",
        element: (
          <CheckAuth>
            <Category />
          </CheckAuth>
        ),
      },
    ],
  },
]);
```

Figure 33 The "routes.js" file

Both the ”Guest.js” and ”CheckAuth.js” files are conditional rendering components that redirect the user based on their authentication state. They achieve this by accessing the ”auth” store using the ”useSelector” hook from Redux. In the ”Guest” component, if the user is not authenticated, it will render the ”children” content, which in this case is the ”Login” or ”Register” page itself. But if authentication state of user is ”true”, it will redirect the user to the homepage using the ”Navigate” component from React Router. The ”replace” prop is set to ”true” which indicates that the user will not be able to go back to the previous page by clicking the back button. In the other hand, the ”CheckAuth” component will let the user

proceed to the ”children content”, which is the ”Home” or ”Category” page, and will send users back to the ”Login” page if authentication state is ”false” at any point during the session.

```
import { useSelector } from "react-redux";
import { Navigate } from "react-router-dom";

export default function Guest({ children }) {
  const auth = useSelector((state) => state.auth);
  return !auth.isAuthenticated ? children : <Navigate to="/" replace={true} />;
}
```

Figure 34 The "Guest.js" file

```
import { useSelector } from "react-redux";
import { Navigate } from "react-router-dom";

export default function CheckAuth({ children }) {
  const auth = useSelector((state) => state.auth);
  return auth.isAuthenticated ? (
    children
  ) : (
    <Navigate to="/login" replace={true} />
  );
}
```

Figure 35 The "CheckAuth.js" file

In the “App.js” file, we fetch the information of the current user after logging in by making a GET request to our server with the endpoint “/user” with our token in the “Authorization header”, which is stored in cookie of our web browser. We then use the “setUser” reducer to update the “user” property of our “auth” store with the data we just obtained from the request. In the “return” section, the navigation bar, which is the “NavBar” component, will always be rendered above the “Outlet” components, which represents all the children of the “App” component.

```
function App() {
  const token = Cookies.get("token");
  const [isLoading, setIsLoading] = useState(true);
  const dispatch = useDispatch();

  async function fetchUser() {
    setIsLoading(true);
    const res = await fetch(`${process.env.REACT_APP_API_URL}/user`, {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    });

    if (res.ok) {
      const user = await res.json();
      dispatch(setUser(user));
    }
    setIsLoading(false);
  }

  useEffect(() => {
    fetchUser();
  }, []);

  if (isLoading) {
    return <p>Loading...</p>;
  }

  return (
    <>
      <NavBar />
      <Outlet />
    </>
  );
}

export default App;
```

Figure 36 The "App.js" file

4.4.3 Navigating in the Front-end

```jsx
export default function NavBar() {
  const dispatch = useDispatch();
  const isAuthenticated = useSelector((state) => state.auth.isAuthenticated);
  const user = useSelector((state) => state.auth.user);
  const navigate = useNavigate();

  function _logOut() {
    Cookies.remove("token");
    dispatch(logOut());
    navigate("/login");
  }

  return (
    <Box sx={{ flexGrow: 1 }}>
      <AppBar style={{ backgroundColor: "green" }} position="static">
        <Toolbar>
          <Typography variant="h6" component="div" sx={{ flexGrow: 1 }}>
            <Link to="/" className="text-white">
              SpendSmartly!
            </Link>
          </Typography>
          {user && isAuthenticated && (
            <Typography sx={{ marginRight: 2 }}>
              <i>{user.firstName} logged in </i>
            </Typography>
          )}
          {isAuthenticated && (
            <Link to="/category" className="text-white">
              <Button color="inherit">Category</Button>
            </Link>
          )}
          {isAuthenticated && (
            <IconButton color="inherit" component="label" onClick={_logOut}>
              <LogoutIcon />
            </IconButton>
          )}
          {!isAuthenticated && (
            <>
              <Link to="/login" className="text-white">
                <Button color="inherit">Login</Button>
              </Link>
              <Link to="/register" className="text-white">
                <Button color="inherit">Register</Button>
              </Link>
            </>
          )}
        </Toolbar>
      </AppBar>
    </Box>
  );
}
```

Figure 37 The "NavBar.js" file

The navigation bar is by default a Basic App Bar component from Material UI with some modifications to meet the requirements of the application. Each option on the navigation bar is wrapped by a ”Link” component from the React Router library, which has an attribute ”to” specifying the destination it takes the user to when being clicked. The logout button is an exception. Instead of using the ”Link” component, it removed the JWT from the browser cookie and use the ”logOut” reducer to empty the user store, set the authentication state to false, and use the ”useNavigate” hook from React Router to send user back to the login page. Our navigation bar also utilizes conditional rendering by accessing the ”isAuthenticated” state deciding to display or hide its children base on the value of the state. As we can see from Figure 37 above, the category, logout button, and the text displaying which user is logged in,

only appear when the authentication value is ”true”. While the login and register button are rendered when the state value is ”false”.

4.4.4 Creating a Transaction

The ”create” function performs an HTTP ”POST” method to the Transaction API from the server, by using the ”fetch” function from JavaScript. The request body is then formed by calling the ”JSON.stringify” method on the ”form” object. Two properties were specified in the ”header” object: ”content-type” which indicates that the request body is in JSON format, and ”Authorization” which contains a Bearer token for authentication.

```
async function create() {
  const res = await fetch(`${process.env.REACT_APP_API_URL}/transaction`, {
    method: "POST",
    body: JSON.stringify(form),
    headers: {
      "content-type": "application/json",
      Authorization: `Bearer ${token}`,
    },
  });

  reload(res);
}
```

Figure 38 The "create" function in the "TransactionForm.js" file

After that, the reload “function” is called which has several callbacks. First, it empties the form by resetting to its initial value. It then sets the “edit” mode of the form to “false” and empties the “editTransaction” value, both of which will be explained later. Then the “fetchTransactions” callback will refresh the transaction list with its newest updates.

```
function reload(res) {
  if (res.ok) {
    setForm(InitialForm);
    setEditMode(false);
    setEditTransaction({});
    fetchTransactions();
  }
}
```

Figure 39 The "reload" function in the "TransactionForm.js" file

4.4.5 Fetching the Transaction List

In our ”Home.js” file, we perform the HTTP “GET” method to the Transaction API to get the latest version of the transaction list, which includes the transaction we just added, modified,

or deleted, and asign the gathered data to the ”transactions” value. We then pass this value to the ”data” props of the ”TransactionList” component, which renders all the transactions to the users.

```
export default function Home() {
  const [transactions, setTransactions] = useState([]);
  const [editTransaction, setEditTransaction] = useState({});

  useEffect(() => {
    fetchTransactions();
  }, []);

  async function fetchTransactions() {
    const token = Cookies.get("token");
    const res = await fetch(`${process.env.REACT_APP_API_URL}/transaction`, {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    });
    const { data } = await res.json();
    setTransactions(data);
  }
```

Figure 40 The "fetchTransactions" function in the "Home.js" file

```
<TransactionList
  data={transactions}
  fetchTransactions={fetchTransactions}
  setEditTransaction={setEditTransaction}
  editTransaction={editTransaction}
/>
```

Figure 41 The "TransactionList" component being called in the "Home.js" file

In the rendering section of the “TransactionList” component, we display the data by mapping the value we just obtained from the “Home” component to all the rows in the table. Each column will represent a property in the “transaction” object: amount, description, category, and date. The last column will contain a delete and edit button.

```
<TableBody>
  {data.map((month) =>
    month.transactions.map((row) => (
      <TableRow
        key={row._id}
        sx={{ "&:last-child td, &:last-child th": { border: 0 } }}
      >
        <TableCell align="center" component="th" scope="row">
          {row.amount} €
        </TableCell>
        <TableCell align="center">{row.description}</TableCell>
        <TableCell align="center" sx={{ fontSize: 20 }}>
          {categoryName(row.category_id)}
        </TableCell>
        <TableCell align="center">{formatDate(row.date)}</TableCell>
        <TableCell align="center">
          <IconButton
            color="warning"
            component="label"
            onClick={() => setEditTransaction(row)}
            disabled={editTransaction.amount !== undefined}
          >
            <EditSharpIcon />
          </IconButton>
          <IconButton
            color="error"
            component="label"
            onClick={() => remove(row._id)}
            disabled={editTransaction.amount !== undefined}
          >
            <DeleteSharpIcon />
          </IconButton>
        </TableCell>
      </TableRow>
    ))
  ))
</TableBody>
```

Figure 42 The data being mapped to the table in the "TransactionList.js" file

4.4.6 Deleting a Transaction

Deleting a transaction requires the client to perform a HTTP ”DELETE” method to the Transaction API. Before sending the request, a confirmation dialog is displayed using ”window.confirm()”. If the user confirms the deletion, the code will continue with the request. Otherwise, no further action will be taken. The structure of the request is the same to both the requests mentioned before, with the exception of an ”_id” parameter being passed in the URL which specifies the ID of the transaction users want to delete. The request is called by pressing the delete button in the table.

```
async function remove(_id) {
  if (!window.confirm("Are you sure you want to delete this transaction?"))
    return;
  const res = await fetch(
    `${process.env.REACT_APP_API_URL}/transaction/${_id}`,
    {
      method: "DELETE",
      headers: {
        Authorization: `Bearer ${token}`,
      },
    }
  );
  if (res.ok) {
    fetchTransactions();
  }
}
```

Figure 43 The "remove" function in the "TransactionList.js" file

```
<IconButton
  color="error"
  component="label"
  onClick={() => remove(row._id)}
  disabled={editTransaction.amount !== undefined}
>
  <DeleteSharpIcon />
</IconButton>
```

Figure 44 The delete button in the "TransactionList.js" file

4.4.7 Editing a Transaction

The ”update” function will perform an HTTP ”PATCH” request to the Transaction API. The code is the same to the ”remove” function.

```
async function update() {
  const res = await fetch(
    `${process.env.REACT_APP_API_URL}/transaction/${editTransaction._id}`,
    {
      method: "PATCH",
      body: JSON.stringify(form),
      headers: {
        "content-type": "application/json",
        Authorization: `Bearer ${token}`,
      },
    }
  );

  reload(res);
}
```

Figure 45 The "update" function in the "TransactionForm.js" file

The transaction form has two mode: ”create” mode (by default) which will create a new transaction, and the ”edit” mode which will allow us to edit a transaction in the table. When we click on the ”edit” button, all the value of the transaction of that row will be transfered to the ”editTransaction” value. Whether the form should ”create” or ”edit” depends on the value ”editMode”. The function ”cancel” will empty the form and the ”editTransaction” value, also set ”editMode” to false, which allow us to create a new transaction again.

```
function handleSubmit(e) {
  e.preventDefault();
  editMode ? update() : create();
}

function handleCancel() {
  setForm(InitialForm);
  setEditMode(false);
  setEditTransaction({});
}
```

Figure 46 The "handleSubmit" and "handleCancel" function in the "TransactionForm.js" file

4.4.8 The Category Page and Its CRUD Methods

Similarly the transaction list, we can operate CRUD methods on the category list also. The code shares a lot of similarities with a few minor tweaks. After perform each request to the Category API, it updates the ”user” state with the latest version of the category list, since ”category” is the property of a user.

```
async function create() {
  const res = await fetch(`${process.env.REACT_APP_API_URL}/category`, {
    method: "POST",
    body: JSON.stringify(form),
    headers: {
      "content-type": "application/json",
      Authorization: `Bearer ${token}`,
    },
  });
  const _user = {
    ...user,
    categories: [...user.categories, { ...form }],
  };
  reload(res, _user);
}

async function update() {
  const res = await fetch(
    `${process.env.REACT_APP_API_URL}/category/${editCategory._id}`,
    {
      method: "PATCH",
      body: JSON.stringify(form),
      headers: {
        "content-type": "application/json",
        Authorization: `Bearer ${token}`,
      },
    }
  );
  const _user = {
    ...user,
    categories: user.categories.map((cat) =>
      cat._id == editCategory._id ? form : cat
    ),
  };
  reload(res, _user);
}
```

Figure 47 The "create" and "update" function in the "CategoryForm.js" file

The ”reload” function for the Category page behaves similarly to one in the Transaction page, which rerenders the page with new data.

```
function reload(res, _user) {
  if (res.ok) {
    setForm(InitialForm);
    setEditMode(false);
    setEditCategory({});
    dispatch(setUser({ user: _user }));
  }
}
```

Figure 48 The "reload" function in the "CategoryForm.js" file

The category list is fetched by making the HTTP “GET” request to the User API, because the category list is attached to each user. The returned data is then asigned to the “user” Redux state.

```
async function fetchUser() {
  setIsLoading(true);
  const res = await fetch(`${process.env.REACT_APP_API_URL}/user`, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  if (res.ok) {
    const user = await res.json();
    dispatch(setUser(user));
  }
  setIsLoading(false);
}
```

Figure 49 The "fetchUser" function in the "App.js" file

Removing a category from the list is also done by making the HTTP “DELETE” request to the Category API. The “user” state is also accessed to be updated with the newest value.

```
async function remove(id) {
  const res = await fetch(`${process.env.REACT_APP_API_URL}/category/${id}`, {
    method: "DELETE",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });
  if (res.ok) {
    const _user = {
      ...user,
      categories: user.categories.filter((cat) => cat._id != id),
    };
    dispatch(setUser({ user: _user }));
  }
}
```

Figure 50 The "remove" function in the "Category.js" file

4.4.9 The Transaction Chart

The ”TransactionChart.js” has a large amount of components imported from the DevExtreme Chart library. The ”data” props are mapped to a new array called ”chartData”. For each item in the data array, the ”month” property is derived from the _id property using the ”dayjs” library. The ”_id” represents the month number (for example, January is 1). The month property is formatted using the ”format()” method to display the month's name.

The component returns a ”Paper” component from Material UI, which acts as a container for the chart. Inside, the ”Chart” component is rendered, which has the chartData passed in the ”data” prop. The ”Chart” component is configured with an ”ArgumentScale” of type ”scaleBand”, which represents the x-axis scale as a band scale. The ”ArgumentAxis” and ”ValueAxis” components represent the x-axis and y-axis respectively. The ”BarSeries” component is utilized to render the bar chart. It has the ”valueField” prop set to "totalExpenses", which specifies the field in each data item that represents the height of the bar. The ”argumentField” is set to "month", which represents the field in each data item that determines the placement of the bar along the x-axis. The ”color” prop is set to ”green”. The ”Animation”, ”EventTracker”, and ”Tooltip” components are also leveraged to provide enhancements to the visual and make chart more interactive.

```
import * as React from "react";
import Paper from "@mui/material/Paper";
import { scaleBand } from "@devexpress/dx-chart-core";
import {
  Chart,
  ArgumentAxis,
  ValueAxis,
  BarSeries,
  Tooltip,
} from "@devexpress/dx-react-chart-material-ui";
import {
  Animation,
  ArgumentScale,
  EventTracker,
} from "@devexpress/dx-react-chart";
import dayjs from "dayjs";

export default function TransactionChart({ data }) {
  const chartData = data.map((item) => {
    item.month = dayjs()
      .month(item._id - 1)
      .format("MMMM");
    return item;
  });

  return (
    <Paper>
      <Chart data={chartData} sx={{ marginTop: 4 }}>
        <ArgumentScale factory={scaleBand} />
        <ArgumentAxis />
        <ValueAxis />
        <BarSeries
          valueField="totalExpenses"
          argumentField="month"
          color="green"
        />
        <Animation />
        <EventTracker />
        <Tooltip />
      </Chart>
    </Paper>
  );
```

Figure 51 The "TransactionChart.js" file

4.4.10 Login and Register

The login process in handled by the ”handleSubmit” function in the ”Login.js” file. the function. The function accesses the input values of the form by creating a new ”FormData” object from the ”event.currentTarget”, which represents the form that triggered the event. The ”form” object is then generated with the ”email” and ”password” fields extracted from the ”data” object using the ”get()” method. The next step is by this point, somewhat familiar as it makes the HTTP ”POST” request to the Auth API. The response from the API, which contains a ”token” and ”user” property, is stored in the ”res” variable.

If the response has an ”ok” status, the function proceeds to set a cookie named "token" with the received token value. It also updates the ”user” state with the value received. It then navigates the user to the "/" route, which is the home page. If the response does not have an ”ok” status, it alerts the user with the "Wrong email or password!" message. The function then returns to prevent any further actions.

```
const handleSubmit = async (event) => {
  event.preventDefault();
  const data = new FormData(event.currentTarget);
  const form = {
    email: data.get("email"),
    password: data.get("password"),
  };
  const res = await fetch(`${process.env.REACT_APP_API_URL}/auth/login`, {
    method: "POST",
    body: JSON.stringify(form),
    headers: {
      "content-type": "application/json",
    },
  });

  const { token, user } = await res.json();

  if (res.ok) {
    Cookie.set("token", token);
    dispatch(setUser(user));
    navigate("/");
  } else {
    alert("Wrong email or password!");
    return;
  }
};
```

Figure 52 The "handleSubmit" function in the "Login.js" file

The register process is also handled by a ”handleSubmit” function in the ”Register.js” file. The code is mostly the same with some modification in the property declaration of the ”form” object. When the ”response” status is ”ok”, it navigates users to the ”/login” route which is the login page or displays an error message if the ”status” is not ”ok”.

```
const handleSubmit = async (event) => {
  event.preventDefault();
  const data = new FormData(event.currentTarget);
  const form = {
    firstName: data.get("firstName"),
    lastName: data.get("lastName"),
    email: data.get("email"),
    password: data.get("password"),
  };
  const res = await fetch(`${process.env.REACT_APP_API_URL}/auth/register`, {
    method: "POST",
    body: JSON.stringify(form),
    headers: {
      "content-type": "application/json",
    },
  });
  if (res.ok) {
    navigate("/login");
  } else {
    alert("Email address is already in the system!");
    return;
  }
};
```

Figure 53 The "hanldeSubmit" function in the "Register.js" file

4.5 Deployment

Throughout the development process, our product can only run locally on the developer's machine. To be accessed publicly by any user, the product needs to be deployed to a public URL using a third-party service. For our project, we use Fly as our server to run the back-end and Netlify to host our front-end. We use two seperate hosting services, instead of only using Fly for the entire codebase. This will prevent us from having to generate a "build" from the client directory every time we make any adjustment to our front-end code. A "build" is a compact version of our code that was translated for the web browser since web browsers cannot understand React.

The third part of the Full Stack Open course from University of Helsinki has detailed information on how to deploy a Node server to Fly (Luukainen, 2023). For our application, after the following the tutorial, we need to run a few additional commands to include the environment variables from our ".env" file. The command is "flyctl secrets set VARIABLE_NAME=VALUE". For example, if we have a value named "PASSWORD" with value "abcd", the command will be "flyctl secrets set PASSWORD=abcd". We need to repeat this command for each environment variables in the ".env" file. To see all the variables set in our Fly application, we can run the command " flyctl secrets list". Figure 54 shows all the "secrets" that are included in our application. All the values were hashed by Fly's algorithm for security.

```
NAME                DIGEST          CREATED AT
JWT_SECRET          b3cac661dd80541e    2023-04-16T09:30:55Z
MONGO_DB_PASSWORD   76ab149de597dd94    2023-04-16T09:33:59Z
MONGO_DB_URL        7fcf330ded549195    2023-04-16T09:35:14Z
MONGO_DB_USERNAME   3cb0f5b92c9417d1    2023-04-16T09:32:50Z
```

Figure 54 All the "secrets" being listed in the Fly application

Deploying the front-end to Netlify is also simply done by following the detailed documentation on their website (Netlify). We connect our Netlify application directly to our GitHub repository with “client” as the base directory since that is where our front-end resides. We use “CI=false npm run build” as our build command. By setting “CI”, which stands for Continuous Integration, to “false”, we disable any CI-specific behavior such as strict checks that could prevent our code from running as intended. The publish directory is “client/build” as our “build” will be generated from the “client” directory. More importantly, we need to set our environment variable, which is “REACT_APP_API_URL”, the URL we obtained after deploying our back-end to Fly.

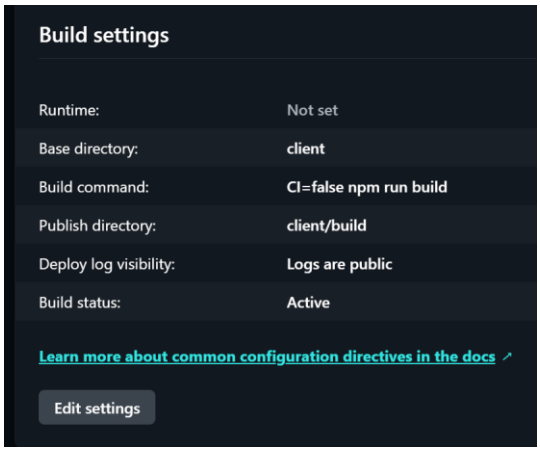


Figure 55 The Build Settings of our Netlify application

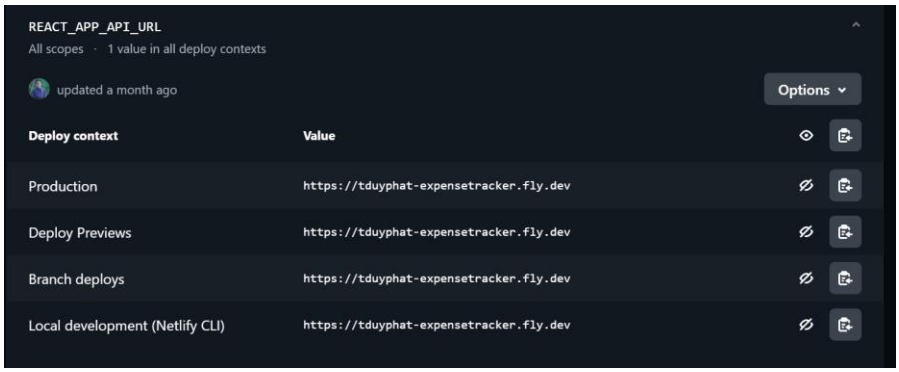


Figure 56 Our Environment Variables in Netlify settings

5 OUTCOME OF THE PROJECT

The application consists of four pages: the login page, register page, home page and category page. The navigation bar always stays on top of every page, and its buttons vary depending on whether user is logged in or not. The application's starting point is the login page.

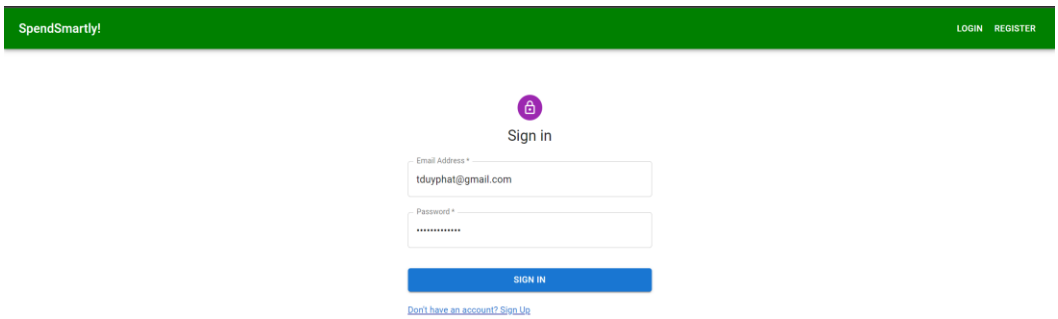


Figure 57 The "Login" screen

Users can register an account by clicking on the link below the "Register" button or the button on the top right corner of the screen. After creating an account, users will be redirected back to the sign in page.

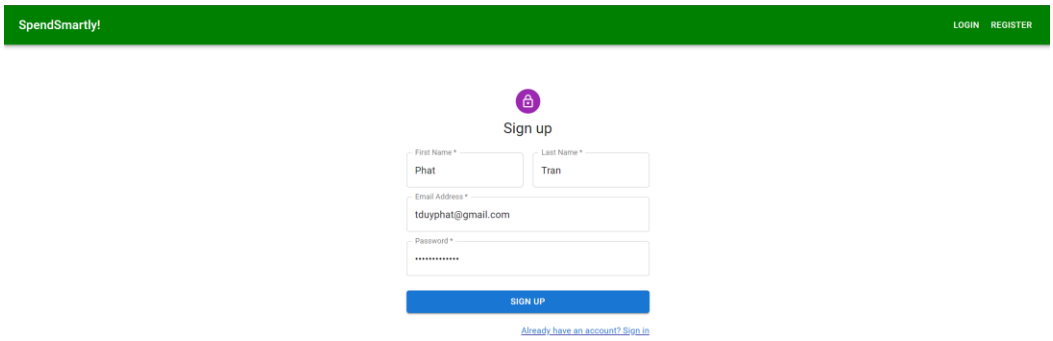


Figure 58 The "Signup" screen

Entering incorrect user credentials during logging in results in an alert that displays "Wrong email or password". Submitting an email address which already exists in the database during

signing up also sends an alert. Both the sign in and sign-up button will be disabled if one of the input fields is left blank.

A successful log-in redirects the user to the home page. The home page contains three components: the transaction form, the transaction list, and the transaction chart. All the transactions submitted to the form will be displayed in the table below and will also affect the bar chart's columns.

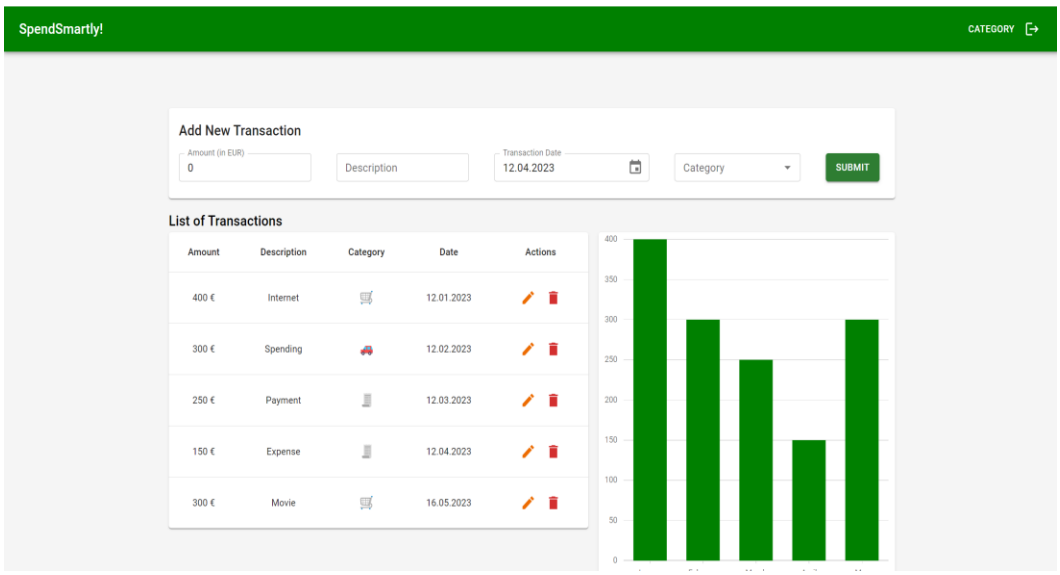


Figure 59 The "Home" screen

The transaction form receives four values: amount of transaction, description, date of transaction and its category. After submitting, each value takes a column in the table below. Each row of transaction also includes an edit button (the pencil shaped one) and a delete button (the trash can). There is a prompt for users to confirm before deleting a transaction. Pressing the edit button on a transaction brings all its value to the form above, changing the form from "create" mode (the default mode) to "edit" mode. Making changes to the value and clicking "update" will update the value of an existing transaction, which then makes the form go back to its initial mode. It is possible to exit the "edit" mode by hitting the "cancel" button.

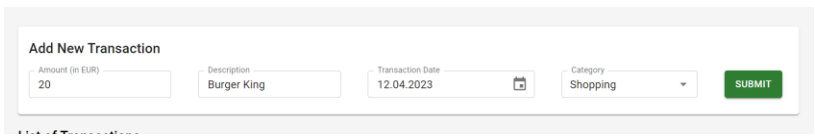


Figure 60 The transaction form in "create" mode

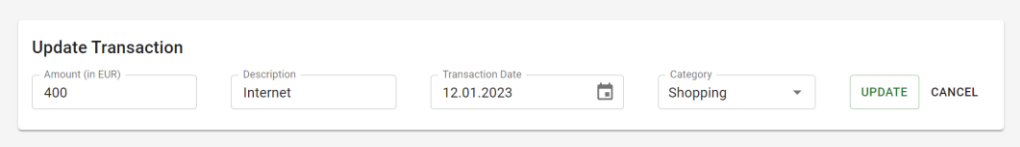


Figure 61 The transaction form in "edit" mode

The chart reacts instantly to all the changes made to the list. It calculates the total amount spent in the month and displays each month as a column.

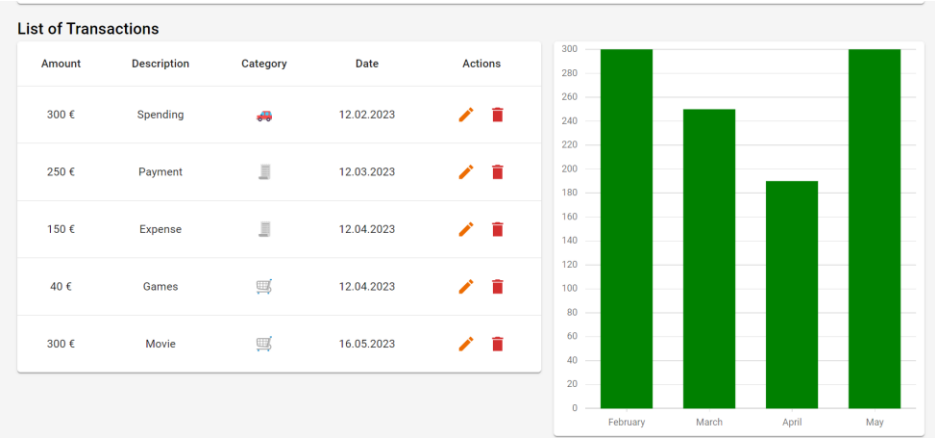


Figure 62 The transaction list and the bar chart

The user is logged in, the navigation bar allows access to the category page. There is a form and a list to customize the category list to the user's liking, including the label and the icon. All the actions on this page behave the same to the transaction page, just with different values. Users can navigate back to the home page or log out at any time.

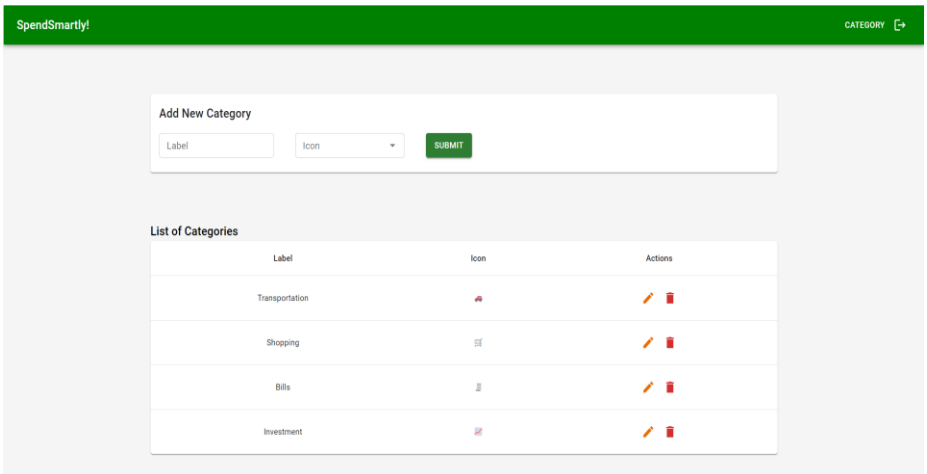


Figure 63 The "Category" page

The product is in its final stage and is ready to use. The URL to the application is:

https://tduyphat-expensetracker.netlify.app/

The GitHub repository of the application can be found at:

https://github.com/tduyphat/MERN-Expense-Tracker-VAMK-Thesis-2023

6 CONCLUSIONS

The thesis is an integral part in the learning path of the author to become a full stack web developer. By putting the time and effort into a project, the author had an opportunity to expand and enhance all coding skills and knowledge gathered throughout four years of study and internship experience. The application has also proved to be useful in daily life as it has given an informative perspective on monthly expense, which encourages the author to adjust the spending habit to make it through the difficult student life.

The MERN stack provides a seamless connection between the front-end and the back-end, provides various methods like fetching data and authenticating the user. The Node and server with Express framework, accompanied by the MongoDB, can handle requests almost instantly and store a huge amount of data without any hiccups. The login and register function also provides a secure experience for the user as the sensitive data such as the transaction list is private to each user. The bar chart represents the total expense of each month, providing a visual appealing touch to the GUI, thus improving the user experience.

The most challenging part by far in the development process was implementing data models from the MongoDB database to the application. Material-UI has saved a huge amount of time spending on designing and position the components in the front-end, giving the author more time to focus on the core functionality of the product.

There are still many potential improvements that can be made on the product in the future to expand the user experience even more. For example, a password recovery system could be helpful in case the users forget the password for their account. The bar chart can be upgraded by having a horizontal navigator to scroll between all the months if there is not enough space in the chart for all the column. Furthermore, a button to sort the transaction list in ascending or descending could also prove to be useful.

REFERENCES

After Academy. (n.d.). Mastering Mongoose for MongoDB and Node.js. Accessed 15.4.2023. https://afteracademy.com/blog/mastering-mongoose-for-mongodb-and-nodejs/.

Luukainen. (2023). Deploying app to internet. Accessed 18.5.2023. https://fullstackopen.com/en/part3/deploying_app_to_internet.

Material UI. (2023). Material UI Documentation. Accessed 14.4.2023. https://mui.com/material-ui/react-button/.

Mongoose. (n.d.). Mongoose Documentation. Accessed 15.4.2023. https://mongoosejs.com/.

Netlify. (n.d.). Netlify Documentation. Accessed 18.5.2023. https://docs.netlify.com/.

Scaler 2023. (n.d.). Scaler 2023. Prop Drilling in React. Accessed 20.4.2023. https://www.scaler.com/topics/react/prop-drilling-in-react/.

Staton. (2022). Students battle to stay afloat in cost of living crisis. Accessed 10.4.2023. https://www.ft.com/content/910c4c1a-df97-49a8-8422-2849d0179da0.

Vailshery. (2022). Most popular web frameworks among developers worldwide 2022. Accessed 14.4.2023. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/.

Wikipedia, Database. (n.d.). Database. Accessed 14.4.2023. https://en.wikipedia.org/wiki/Database.

Wikipedia, npm. (n.d.). npm (software). Accessed 14.4.2023. https://en.wikipedia.org/wiki/Npm_(software).

Workfall. (2022). How to build and deploy a MERN Stack Application on AWS?. Accessed 14.4.2023. https://www.workfall.com/learning/blog/how-to-build-and-deploy-a-mern-stack-application-on-aws/.