**Budget Tracking Application Technical Documentation**

**Introduction**

The Budget Tracking Application is a web-based tool designed to help users manage their monthly budgets effectively. It allows users to track their expenses, set budget limits, and visualize their spending trends over time. The application features user authentication, budget entry management, filtering by dates, and reporting capabilities.

**Tech Stack**

The application is built using the following technologies:

* **Frontend:**
  + React: A JavaScript library for building user interfaces.
* **Backend:**
  + Node.js: A JavaScript runtime environment for server-side development.
  + Express: A minimalist web framework for Node.js.
* **Database:**
  + Mongo DB
  + Mongoose

**Features**

**User Authentication/Authorization**

* **Sign In:** Users can sign in to their accounts using their email and password.
* **Sign Up:** New users can create an account by providing their email, name, password, and budget limit.
* **JWT-Based Authentication:** Token-based authentication is implemented using JSON Web Tokens (JWT) for secure authentication and authorization.

**Budget Entry Management**

* **Add Entry:** Users can add new budget entries, specifying the date/time, name of transaction/purchase, and price/budget.
* **List Entries:** Users can view a listing of all their budget entries.
* **Edit Entry:** Users can edit existing budget entries to update the transaction details.
* **Delete Entry:** Users can delete budget entries that are no longer needed.

**Budget Limit Notification**

* **Exceeded Budget Notification:** Users receive notifications or indications when they exceed their budget limit for the current month.

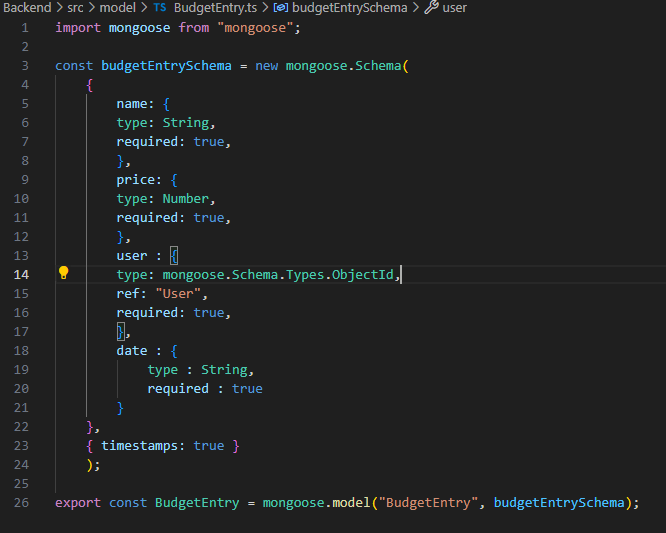
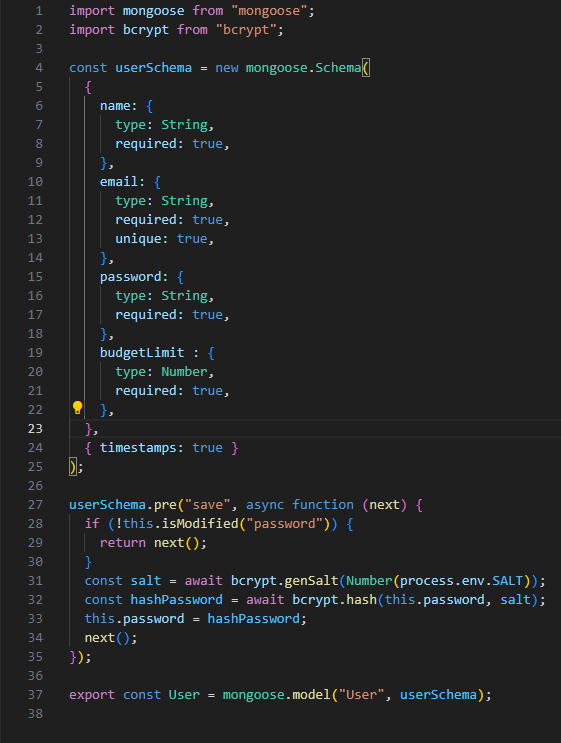
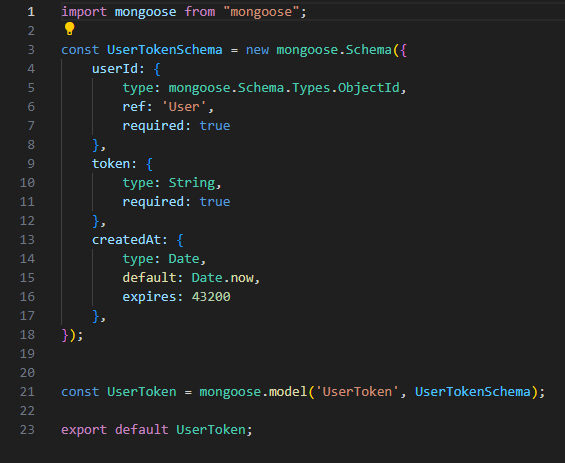
**Filtering by Dates**

* **Date Filtering:** Users can filter budget entries using a calendar or date picker. By default, the current date is selected.

**Reporting**

* **Budget Trend Visualization:** Users can view their budget trends for different time periods, including the last month, last 6 months, and last 12 months.
* **Exceeded Limit Indicator:** The budget trend chart highlights instances where users have exceeded their budget limit.

**Database Schema**

* **Budget Entry Schema: **
* **User Schema **
* **UserToken Schema **

**Routes Documentation**

**Authentication Routes**

**POST /auth**

* **Description: Endpoint for user login.**
* **Request Body:**
  + **email (string): User's email address.**
  + **password (string): User's password.**
* **Middleware:**
  + **validate: Validates the login schema.**
* **Controller:**
  + **login: Logs in the user.**
* **Response:**
  + **200 OK: Returns user data and access token upon successful login.**
  + **400 Bad Request: Returns validation error if request body is invalid.**
  + **401 Unauthorized: Returns error message if login credentials are incorrect.**

**DELETE /auth**

* **Description: Endpoint for user logout.**
* **Request Body:**
  + **refreshToken (string): User's refresh token.**
* **Middleware:**
  + **validate: Validates the refresh token schema.**
* **Controller:**
  + **logout: Logs out the user.**
* **Response:**
  + **204 No Content: Returns no content upon successful logout.**
  + **400 Bad Request: Returns validation error if request body is invalid.**
  + **401 Unauthorized: Returns error message if refresh token is invalid or expired.**

**POST /auth/refresh**

* **Description: Endpoint for refreshing access token.**
* **Request Body:**
  + **refreshToken (string): User's refresh token.**
* **Middleware:**
  + **validate: Validates the refresh token schema.**
* **Controller:**
  + **refreshToken: Refreshes user's access token.**
* **Response:**
  + **200 OK: Returns new access token upon successful token refresh.**
  + **400 Bad Request: Returns validation error if request body is invalid.**
  + **401 Unauthorized: Returns error message if refresh token is invalid or expired.**

**User Routes**

**GET /user**

* **Description: Endpoint to get all users.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
* **Controller:**
  + **getUsers: Retrieves all users.**
* **Response:**
  + **200 OK: Returns list of users.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**POST /user**

* **Description: Endpoint to create a new user.**
* **Request Body:**
  + **email (string): User's email address.**
  + **name (string): User's name.**
  + **password (string): User's password.**
  + **budgetLimit (number): User's budget limit.**
* **Middleware:**
  + **validate: Validates the user creation schema.**
* **Controller:**
  + **createUser: Creates a new user.**
* **Response:**
  + **201 Created: Returns newly created user data.**
  + **400 Bad Request: Returns validation error if request body is invalid.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**GET /user/:id**

* **Description: Endpoint to get user by ID.**
* **Request Params:**
  + **id (string): User ID.**
* **Middleware:**
  + **validate: Validates the user ID schema.**
* **Controller:**
  + **getUserById: Retrieves user by ID.**
* **Response:**
  + **200 OK: Returns user data.**
  + **400 Bad Request: Returns validation error if request parameter is invalid.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**Budget Routes**

**GET /budget**

* **Description: Endpoint to get user's budget entries.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
* **Controller:**
  + **getUserBudgets: Retrieves user's budget entries.**
* **Response:**
  + **200 OK: Returns list of user's budget entries.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**POST /budget**

* **Description: Endpoint to add a new budget entry.**
* **Request Body:**
  + **date (string): Date of the budget entry.**
  + **name (string): Name of transaction/purchase.**
  + **price (number): Price/budget amount.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
  + **validate: Validates the add budget entry schema.**
* **Controller:**
  + **addBudgetEntry: Adds a new budget entry.**
* **Response:**
  + **201 Created: Returns newly created budget entry.**
  + **400 Bad Request: Returns validation error if request body is invalid.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**GET /budget/date**

* **Description: Endpoint to get user's budget entries for last month.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
* **Controller:**
  + **getBudgetForLastMonth: Retrieves user's budget entries for last month.**
* **Response:**
  + **200 OK: Returns list of user's budget entries for last month.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**DELETE /budget/:id**

* **Description: Endpoint to delete a budget entry by ID.**
* **Request Params:**
  + **id (string): Budget entry ID.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
* **Controller:**
  + **deleteBudgetEntry: Deletes budget entry by ID.**
* **Response:**
  + **204 No Content: Returns no content upon successful deletion.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**GET /budget/:date**

* **Description: Endpoint to get user's budget entries for a specific date.**
* **Request Params:**
  + **date (string): Date to filter budget entries.**
* **Middleware:**
  + **isAuthenticated: Authenticates user.**
* **Controller:**
  + **getBudgetsByDate: Retrieves user's budget entries for the specified date.**
* **Response:**
  + **200 OK: Returns list of user's budget entries for the specified date.**
  + **401 Unauthorized: Returns error message if user is not authenticated.**

**Using Sequelize instead of mongoose for an SQL ORM.**

* + 1. **Install Sequlize:** Use the command npm i sequlize
    2. **You’ll also have to manually install the driver for your database of choice. As I am using Postgres, I am using the following command:** npm install --save pg pg-hstore # Postgres

**Create a Global instance of sequlize which connects to your DB:**

import { Sequelize } from 'sequelize';

import dotenv from 'dotenv';

dotenv.config();

const sequelize = new Sequelize(

  process.env.DB\_NAME,

  process.env.DB\_USER,

  process.env.DB\_PASSWORD, {

  host: process.env.DB\_HOST,

  dialect: "postgres"

});

export default sequelize;

**After Creating the global instance use it to create models like so:**

**Budget Entry Model:**

import { Model, DataTypes, Optional } from 'sequelize';

import sequelize from '../../config/Sequelize';

// These interfaces are for TypeScript typings for the model attributes and instance

interface BudgetEntryAttributes {

  id?: number;

  name: string;

  price: number;

  userId: number;

  date: string;

}

interface BudgetEntryCreationAttributes extends Optional<BudgetEntryAttributes, 'id'> {}

export class BudgetEntry extends Model<BudgetEntryAttributes, BudgetEntryCreationAttributes> implements BudgetEntryAttributes {

  public id!: number;

  public name!: string;

  public price!: number;

  public userId!: number;

  public date!: string;

  public readonly createdAt!: Date;

  public readonly updatedAt!: Date;

}

BudgetEntry.init({

  id: {

    type: DataTypes.INTEGER,

    autoIncrement: true,

    primaryKey: true,

  },

  name: {

    type: DataTypes.STRING,

    allowNull: false

  },

  price: {

    type: DataTypes.DECIMAL(10, 2), // Adjust precision and scale according to your requirements

    allowNull: false

  },

  userId: {

    type: DataTypes.INTEGER,

    allowNull: false,

    references: {

      model: 'Users', // Adjust according to your table naming conventions

      key: 'id'

    }

  },

  date: {

    type: DataTypes.STRING,

    allowNull: false

  }

}, {

  sequelize,

  modelName: 'BudgetEntry',

  timestamps: true

});

**User Model:**

import { Model, DataTypes, Optional } from 'sequelize';

import bcrypt from 'bcrypt';

import sequelize from '../../config/Sequelize';

interface UserAttributes {

  id?: number;

  name: string;

  email: string;

  password: string;

  budgetLimit: number;

}

interface UserCreationAttributes extends Optional<UserAttributes, 'id'> {}

export class User extends Model<UserAttributes, UserCreationAttributes> implements UserAttributes {

  public id!: number;

  public name!: string;

  public email!: string;

  public password!: string;

  public budgetLimit!: number;

  // timestamps!

  public readonly createdAt!: Date;

  public readonly updatedAt!: Date;

}

User.init({

  id: {

    type: DataTypes.INTEGER,

    autoIncrement: true,

    primaryKey: true,

  },

  name: {

    type: DataTypes.STRING,

    allowNull: false

  },

  email: {

    type: DataTypes.STRING,

    allowNull: false,

    unique: true

  },

  password: {

    type: DataTypes.STRING,

    allowNull: false

  },

  budgetLimit: {

    type: DataTypes.DECIMAL(10, 2), // Adjust precision and scale according to your requirements

    allowNull: false

  }

}, {

  sequelize,

  modelName: 'User',

  timestamps: true,

  hooks: {

    beforeSave: async (user: User) => {

      if (user.changed('password')) {

        const salt = await bcrypt.genSalt(Number(process.env.SALT));

        user.password = await bcrypt.hash(user.password, salt);

      }

    }

  }

});

**Connecting and syncing your Sequlize with the database.**

**After You have created the model now it’s time to sync with the database and make the tables in the database:**

In your index.ts File :

sequelize.authenticate()

  .then(() => console.log('Database connected.'))

  .catch(err => console.error('Unable to connect to the database:', err));

(async () => {

  await User.sync({ force: true });

  await UserToken.sync({ force: true });

  await BudgetEntry.sync({ force: true });

})();

**Sequlize.authenticate Function checks if you are connected to the database or not.**

**[Model].Sync({force : true}); makes the tables in the database according to you model in your code.**

**Using TypeORM instead of mongoose:**

* + 1. Install Type ORM With the command npm install typeorm –save
    2. You will need to install reflect-metadata shim using the command

npm install reflect-metadata –save

* + 1. Import it somewhere in the global place of your app (for example index.ts)
       - import "reflect-metadata";
    2. You may need to install node typings with the command

npm install @types/node –save-dev

* + 1. Install a database driver I am using postgres so for it you will use the command

npm install pg --save

* + 1. For mysql use npm install mysql2 –save
    2. You will also need to change TypeScript Configuration
       - In your tsconfig.json file add these two lines

{

    "compilerOptions": {

        "module": "CommonJS",

        "esModuleInterop": true,

        "target": "es2021",

        "moduleResolution": "node",

        "sourceMap": true,

        "outDir": "dist",

        "rootDir": "src",

        "emitDecoratorMetadata": true , // this

        "experimentalDecorators":true   // this

    },

    "include": ["src/\*\*/\*"],

    "exclude": ["dist"]

}

* + 1. Now You are ready to use TypeORM.

**Creating models :**

* + 1. **User model:**

import { Entity, PrimaryGeneratedColumn, Column, BeforeInsert, CreateDateColumn, UpdateDateColumn, OneToMany, BaseEntity } from 'typeorm';

import bcrypt from 'bcrypt';

import { BudgetEntry } from './BudgetEntry';

@Entity()

export class User extends BaseEntity {

    @PrimaryGeneratedColumn()

    id: number;

    @Column({ type: "varchar", length: 255, nullable: false })

    name: string;

    @Column({ type: "varchar", length: 255, nullable: false, unique: true })

    email: string;

    @Column({ type: "varchar", nullable: false , select: false})

    password: string;

    @Column({ type: "float", nullable: false })

    budgetLimit: number;

    @CreateDateColumn()

    createdAt: Date;

    @UpdateDateColumn()

    updatedAt: Date;

    @OneToMany(() => BudgetEntry, budgetEntry => budgetEntry.user)

    budgetEntries: BudgetEntry[];

    @BeforeInsert()

    async hashPassword() {

        const salt = await bcrypt.genSalt(Number(process.env.SALT || 10));  // Default to 10 if SALT is not defined

        this.password = await bcrypt.hash(this.password, salt);

    }

}

* + 1. **BudgetEntry Model:**

import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, CreateDateColumn, UpdateDateColumn, BaseEntity } from 'typeorm';

import { User } from './User';  // Assuming you have a User entity defined elsewhere

@Entity()

export class BudgetEntry extends BaseEntity{

    @PrimaryGeneratedColumn()

    id: number;

    @Column({ type: "varchar", length: 255, nullable: false })

    name: string;

    @Column({ type: "float", nullable: false })

    price: number;

    @ManyToOne(() => User, user => user.budgetEntries, { nullable: false })

    user: User;

    @Column({ type: "varchar", nullable: false })

    date: string;

    @CreateDateColumn()

    createdAt: Date;

    @UpdateDateColumn()

    updatedAt: Date;

}

**Make A function in your conifg directory to connect to the Db and call it in the index.ts file :**

import "reflect-metadata"

import { DataSource } from "typeorm"

import { BudgetEntry } from "../model/BudgetEntry";

import { User } from "../model/User";

import { UserToken } from "../model/UserToken";

const connectDB = async () => {

    const AppDataSource = new DataSource({

        type: "postgres",

        host: "localhost",

        port: 5432,

        username: "postgres",

        password: "1234",

        database: "TestTypeOrm",

        entities: [BudgetEntry, User, UserToken],

        synchronize: true,

        logging: true,

    });

    try {

        await AppDataSource.initialize();

        console.log("Database connected")

    } catch (error){

        console.log(error)

        console.log("Error connecting to database")

    }

}

export default connectDB;

**Now Will be able to perform crud operation in your APIs.**