Sure, here's a proposed index for your project report:

Expense Tracker Project Report

Table of Contents

1. Introduction
2. Project Motivation
3. Project Overview
4. Backend Development
    1. Setup and Configuration
    2. Database Design
    3. Spring Boot Application
    4. Security Implementation
    5. REST API Development
5. Frontend Development
    1. Setup and Configuration
    2. UI Design
    3. React Components
    4. State Management
    5. Integration with Backend
6. Testing
7. Deployment
8. Conclusion
9. Future Improvements
10. Errors Encountered During Development

Introduction

The Expense Tracker Project is a web application designed to help users, particularly students studying abroad, manage their expenses effectively. As an Erasmus student, one of the biggest challenges is keeping track of various expenses incurred during the course of the study program. This application aims to provide a simple yet comprehensive solution to record, manage, and analyze personal expenditures.

Project Motivation

The project leverages modern web technologies, with a backend built using Spring Boot and PostgreSQL, and a frontend developed using React. The primary objective is to create a user-friendly interface that allows users to register, log in, and manage their expense categories and transactions. Additionally, the application ensures secure data handling and user authentication to protect sensitive information.

Project Overview

By using this expense tracker, users can categorize their expenses, set budgets, and view detailed reports of their spending habits. This helps in making informed financial decisions, planning budgets effectively, and ultimately, reducing financial stress. The project not only serves as a practical tool for personal finance management but also provides an opportunity to gain hands-on experience in full-stack web development.

Backend Development (Spring Boot and PostgreSQL)

The backend of the Expense Tracker Project is built using Spring Boot, a powerful framework for developing Java-based applications. The database used is PostgreSQL, which is a reliable and robust relational database management system. The backend is responsible for handling user authentication, managing expense categories and transactions, and ensuring secure data storage and retrieval.

Step 1: Setting Up the Development Environment

1. **Install Java Development Kit (JDK)**: Ensure that JDK is installed on your system. Spring Boot requires Java to run.
2. **Install Spring Boot**: Set up Spring Boot using the Spring Initializr (https://start.spring.io/) by selecting dependencies like Spring Web, Spring Data JPA, and PostgreSQL.
3. **Install PostgreSQL**: Download and install PostgreSQL. Create a new database for the application.

Step 2: Project Structure and Configuration

1. **Create the Project**: Generate the Spring Boot project using the Spring Initializr and import it into your preferred Integrated Development Environment (IDE) like IntelliJ IDEA or Eclipse.

2. **Configure Application Properties**: Set up the database connection in `application.properties` file.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/expense_tracker_db
spring.datasource.username=your_postgres_username
spring.datasource.password=your_postgres_password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Step 3: Implementing the Domain Model

1. **Create Domain Classes**: Define the domain classes representing the data models, such as User, Category, and Transaction.

```
public class User {
    private Integer userId;
    private String firstName;
    private String lastName;
    private String email;
    private String password;
    // Getters and setters
}

public class Category {
    private Integer categoryId;
    private Integer userId;
    private String title;
    private String description;
    private Double totalExpense;
    // Getters and setters
}

public class Transaction {
    private Integer transactionId;
    private Integer categoryId;
    private Integer userId;
    private Double amount;
    private String note;
    private Long transactionDate;
    // Getters and setters
}
```

Step 4: Creating the Repository Layer

1. **Define Repository Interfaces**: Create interfaces for data access operations. These interfaces will use Spring Data JPA to interact with the database.

```
public interface UserRepository {
    Integer create(String firstName, String lastName, String email,
String password) throws EtAuthException;
    User findByEmailAndPassword(String email, String password) throws
EtAuthException;
    Integer getCountByEmail(String email);
    User findById(Integer userId);
}

public interface CategoryRepository {
    List<Category> findAll(Integer userId) throws
EtResourceNotFoundException;
    Category findById(Integer userId, Integer categoryId) throws
EtResourceNotFoundException;
    Integer create(Integer userId, String title, String description)
throws EtBadRequestException;
    void update(Integer userId, Integer categoryId, Category category)
throws EtBadRequestException;
    void removeById(Integer userId, Integer categoryId);
}

public interface TransactionRepository {
    List<Transaction> findAll(Integer userId, Integer categoryId);
    Transaction findById(Integer userId, Integer categoryId, Integer
transactionId) throws EtResourceNotFoundException;
    Integer create(Integer userId, Integer categoryId, Double amount,
String note, Long transactionDate) throws EtBadRequestException;
    void update(Integer userId, Integer categoryId, Integer
transactionId, Transaction transaction) throws EtBadRequestException;
    void removeById(Integer userId, Integer categoryId, Integer
transactionId) throws EtResourceNotFoundException;
}
```

Step 5: Implementing the Service Layer

1. **Create Service Interfaces and Implementations**: Define the business logic in service interfaces and their implementations.

```
public interface UserService {
    User validateUser(String email, String password) throws
EtAuthException;
    User registerUser(String firstName, String lastName, String email,
String password) throws EtAuthException;
}

@Service
@Transactional
public class UserServiceImpl implements UserService {
    @Autowired
    UserRepository userRepository;

    @Override
    public User validateUser(String email, String password) throws
EtAuthException {
        if(email != null) email = email.toLowerCase();
        return userRepository.findByEmailAndPassword(email, password);
    }

    @Override
    public User registerUser(String firstName, String lastName, String
email, String password) throws EtAuthException {
        Pattern pattern = Pattern.compile("^(.+)@(.+)$");
        if(email != null) email = email.toLowerCase();
        if(!pattern.matcher(email).matches())
            throw new EtAuthException("Invalid email format");
        Integer count = userRepository.getCountByEmail(email);
        if(count > 0)
            throw new EtAuthException("Email already in use");
        Integer userId = userRepository.create(firstName, lastName,
email, password);
        return userRepository.findById(userId);
    }
}
```

Step 6: Implementing the Controller Layer

1. **Create REST Controllers**: Define the RESTful endpoints for handling HTTP requests.

```
@RestController
@RequestMapping("/api/users")
public class UserResource {
    @Autowired
    UserService userService;

    @PostMapping("/login")
    public ResponseEntity<Map<String, String>> loginUser(@RequestBody
Map<String, Object> userMap) {
        String email = (String) userMap.get("email");
        String password = (String) userMap.get("password");
        User user = userService.validateUser(email, password);
        return new ResponseEntity<>(generateJWTToken(user),
HttpStatus.OK);
    }

    @PostMapping("/register")
    public ResponseEntity<Map<String, String>>
registerUser(@RequestBody Map<String, Object> userMap) {
        String firstName = (String) userMap.get("firstName");
        String lastName = (String) userMap.get("lastName");
        String email = (String) userMap.get("email");
        String password = (String) userMap.get("password");
        User user = userService.registerUser(firstName, lastName,
email, password);
        return new ResponseEntity<>(generateJWTToken(user),
HttpStatus.OK);
    }

    private Map<String, String> generateJWTToken(User user) {
        long timestamp = System.currentTimeMillis();
        String token =
Jwts.builder().signWith(SignatureAlgorithm.HS256,
Constants.API_SECRET_KEY)
                .setIssuedAt(new Date(timestamp))
                .setExpiration(new Date(timestamp +
Constants.TOKEN_VALIDITY))
                .claim("userId", user.getUserId())
                .claim("email", user.getEmail())
                .claim("firstName", user.getFirstName())
                .claim("lastName", user.getLastName())
                .compact();
        Map<String, String> map = new HashMap<>();
        map.put("token", token);
        return map;
    }
}
```

Step 7: Security Configuration and CORS

1. **Configure Security**: Set up security configurations to handle user authentication and authorization.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .cors().configurationSource(corsConfigurationSource())
                .and()
                .csrf().disable()
                .authorizeRequests()
                .antMatchers("/**").permitAll();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://localhost:3000");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);
        return source;
    }
}
```

2. **Configure CORS**: Allow cross-origin requests from the frontend.

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET", POST", "PUT", "DELETE",
"OPTIONS")
                .allowedHeaders("*")
                .allowCredentials(true);
    }
}
```

By following these steps, the backend of the Expense Tracker Project is set up to handle user authentication, manage categories, and transactions, and provide a secure API for the frontend to interact with. The next section will cover the frontend development using React.

Frontend Development (React)

The frontend of the Expense Tracker Project is built using React, a popular JavaScript library for building user interfaces. The frontend interacts with the backend API to perform CRUD operations on categories and transactions, and to handle user authentication.

Step 1: Setting Up the Development Environment

1. **Install Node.js and npm**: Ensure that Node.js and npm (Node Package Manager) are installed on your system.
2. **Create a React App**: Use Create React App to set up the project. This tool sets up a modern web development environment with a single command.

```
npx create-react-app expense-tracker
cd expense-tracker
```

Step 2: Project Structure and Initial Setup

1. **Install Required Packages**: Install additional packages like `axios` for making HTTP requests and `react-router-dom` for routing.

```
npm install axios react-router-dom
```

2. **Set Up Routing**: Configure routing in the `App.js` file to handle navigation between different pages.

```
import React, { useState, useEffect } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Login from './Login';
import Register from './Register';
import Dashboard from './Dashboard';
import CategoryTransactions from './CategoryTransactions';
import EditCategory from './EditCategory';
import PrivateRoute from '../utils/PrivateRoute';
import Header from './Header';

import '../styles.css';

const App = () => {
    const [isLoggedIn, setIsLoggedIn] = useState(false);

    useEffect(() => {
        const token = localStorage.getItem('token');
        setIsLoggedIn(!!token);
    }, []);

    const handleLogout = () => {
        localStorage.removeItem('token');
        setIsLoggedIn(false);
    };

    return (
        <Router>
            <Header isLoggedIn={isLoggedIn} handleLogout=
{handleLogout} />
            <div className="container">
                <Routes>
                    <Route path="/login" element={<Login />} />
                    <Route path="/register" element={<Register />} />
                    <Route path="/dashboard" element={
                        <PrivateRoute>
                            <Dashboard />
                        </PrivateRoute>
                    } />
                    <Route path="/categories/:categoryId" element={
                        <PrivateRoute>
                            <CategoryTransactions />
                        </PrivateRoute>
                    } />
                    <Route path="/edit-category/:categoryId" element={
                        <PrivateRoute>
                            <EditCategory />
                        </PrivateRoute>
                    } />
                </Routes>
            </div>
        </Router>
    );
};

export default App;
```

Step 3: Implementing Authentication

1. **Login Component**: Create a login form to authenticate users.

```
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { login } from '../services/authService';

const Login = () => {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const navigate = useNavigate();

    const handleLogin = async (event) => {
        event.preventDefault();
        try {
            const data = await login(email, password);
            localStorage.setItem('token', data.token);
            navigate('/dashboard');
        } catch (error) {
            alert('Login failed: ' + error.message);
        }
    };

    return (
        <form onSubmit={handleLogin}>
            <h2>Login</h2>
            <input
                type="email"
                placeholder="Email"
                value={email}
                onChange={(e) => setEmail(e.target.value)}
                required
                autoComplete="email"
            />
            <input
                type="password"
                placeholder="Password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
                required
                autoComplete="current-password"
            />
            <button type="submit">Login</button>
        </form>
    );
};

export default Login;
```

2. **Register Component**: Create a registration form for new users.

```
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { register } from '../services/authService';

const Register = () => {
    const [firstName, setFirstName] = useState('');
    const [lastName, setLastName] = useState('');
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const navigate = useNavigate();

    const handleRegister = async (event) => {
        event.preventDefault();
        try {
            await register(firstName, lastName, email, password);
            alert('Registration successful');
            navigate('/login');
        } catch (error) {
            alert('Registration failed: ' + error.message);
        }
    };

    return (
        <form onSubmit={handleRegister}>
            <h2>Register</h2>
            <input
                type="text"
                placeholder="First Name"
                value={firstName}
                onChange={(e) => setFirstName(e.target.value)}
                required
                autoComplete="given-name"
            />
            <input
                type="text"
                placeholder="Last Name"
                value={lastName}
                onChange={(e) => setLastName(e.target.value)}
                required
                autoComplete="family-name"
            />
            <input
                type="email"
                placeholder="Email"
                value={email}
                onChange={(e) => setEmail(e.target.value)}
                required
                autoComplete="email"
            />
            <input
                type="password"
                placeholder="Password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
                required
                autoComplete="new-password"
            />
            <button type="submit">Register</button>
        </form>
    );
};

export default Register;
```

3. **Authentication Service**: Define functions to handle API requests for login and registration.

```
import axios from 'axios';

const API_URL = 'http://localhost:8080/api';
```

```
export const login = async (email, password) => {
    const response = await axios.post(`${API_URL}/users/login`, {
email, password });
    return response.data;
};

export const register = async (firstName, lastName, email, password)
=> {
    const response = await axios.post(`${API_URL}/users/register`, {
firstName, lastName, email, password });
    return response.data;
};
```

Step 4: Managing Categories

1. **Dashboard Component**: Create a dashboard to display and manage expense categories.

```
import React, { useState, useEffect } from 'react';
import { Link, useNavigate } from 'react-router-dom';
import { getCategories, deleteCategory, createCategory } from
'../services/categoryService';

const Dashboard = () => {
    const [categories, setCategories] = useState([]);
    const [title, setTitle] = useState('');
    const [description, setDescription] = useState('');
    const navigate = useNavigate();

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

    const fetchCategories = async () => {
        try {
            const data = await getCategories();
            setCategories(data);
        } catch (error) {
            console.error('Failed to fetch categories', error);
        }
    };

    const handleDelete = async (categoryId) => {
        try {
            await deleteCategory(categoryId);
            fetchCategories(); // Refresh categories after deletion
        } catch (error) {
            console.error('Failed to delete category', error);
        }
    };
    const handleAddCategory = async () => {
        try {
            await createCategory(title, description);
            setTitle('');
            setDescription('');
            fetchCategories(); // Refresh categories after adding
        } catch (error) {
            console.error('Failed to add category', error);
        }
    };

    return (
        <div>
            <h1>Dashboard</h1>
            <div className="categories">
                <h2>Categories</h2>
                <ul>
                    {categories.map(category => (
                        <li key={category.categoryId}>
                            <Link to=
{`/categories/${category.categoryId}`}>
                                {category.title}
                            </Link>
                            <button onClick={() =>
handleDelete(category.categoryId)}>Delete</button>
                            <button onClick={() => navigate(`/edit-
category/${category.categoryId}`)}>Edit</button>
                        </li>
                    ))}
                </ul>
                <h3>Add Category</h3>
                <input
                    type="text"
                    placeholder="Title"
                    value={title}
                    onChange={(e) => setTitle(e.target.value)}
                />
                <input
                    type="text"
                    placeholder="Description"
                    value={description}
                    onChange={(e) => setDescription(e.target.value)}
                />
                <button onClick={handleAddCategory}>Add
Category</button>
            </div>
        </div>
    );
};

export default Dashboard;
```

2. **Category Service**: Define functions to handle API requests for categories.

```
import axios from 'axios';

const API_URL = 'http://localhost:8080/api/categories';

export const getCategories = async () => {
    const response = await axios.get(API_URL, {
        headers: {
            Authorization: `Bearer ${localStorage.getItem('token')}`
```

} }); return response.data; };

```
export const deleteCategory = async (categoryId) => {
    await axios.delete(`${API_URL}/${categoryId}`, {
        headers: {
            Authorization: `Bearer ${localStorage.getItem('token')}`
        }
    });
};

export const createCategory = async (title, description) => {
    const response = await axios.post(API_URL, {
        title,
        description
    }, {
        headers: {
            'Content-Type': 'application/json',
            Authorization: `Bearer ${localStorage.getItem('token')}`
        }
    });
    return response.data;
};

export const getCategory = async (categoryId) => {
    const response = await axios.get(`${API_URL}/${categoryId}`, {
        headers: {
            Authorization: `Bearer ${localStorage.getItem('token')}`
        }
    });
    return response.data;
};

export const updateCategory = async (categoryId, title, description) => {
    await axios.put(`${API_URL}/${categoryId}`, {
        title,
        description
    }, {
        headers: {
            'Content-Type': 'application/json',
```

~~~
            Authorization: `Bearer ${localStorage.getItem('token')}`
        }
    });
};
```
~~~

3. **Edit Category Component**: Create a form to edit existing categories.

```
import React, { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';
import { getCategory, updateCategory } from
'../services/categoryService';

const EditCategory = () => {
    const { categoryId } = useParams();
    const [title, setTitle] = useState('');
    const [description, setDescription] = useState('');
    const navigate = useNavigate();

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

    const fetchCategory = async () => {
        try {
            const category = await getCategory(categoryId);
            setTitle(category.title);
            setDescription(category.description);
        } catch (error) {
            console.error('Failed to fetch category', error);
        }
    };

    const handleUpdate = async () => {
        try {
            await updateCategory(categoryId, title, description);
            navigate('/dashboard');
        } catch (error) {
            console.error('Failed to update category', error);
        }
    };

    return (
        <div>
            <h1>Edit Category</h1>
            <input
                type="text"
                placeholder="Title"
                value={title}
                onChange={(e) => setTitle(e.target.value)}
            />
            <input
```

```
                type="text"
                placeholder="Description"
                value={description}
                onChange={(e) => setDescription(e.target.value)}
            />
            <button onClick={handleUpdate}>Update Category</button>
        </div>
    );
};

export default EditCategory;
```

Step 5: Managing Transactions

1. **Category Transactions Component**: Display and manage transactions for a specific category.

```
import React, { useEffect, useState } from 'react';
import { useParams } from 'react-router-dom';
import { getTransactions, createTransaction, deleteTransaction,
updateTransaction } from '../services/transactionService';

const CategoryTransactions = () => {
    const { categoryId } = useParams();
    const [transactions, setTransactions] = useState([]);
    const [newTransaction, setNewTransaction] = useState({ amount: '',
note: '', transactionDate: '' });
    const [editMode, setEditMode] = useState(false);
    const [currentTransaction, setCurrentTransaction] =
useState(null);

    useEffect(() => {
        const fetchTransactions = async () => {
            try {
                const data = await getTransactions(categoryId);
                setTransactions(data);
            } catch (error) {
                console.error(error);
            }
        };

        fetchTransactions();
    }, [categoryId]);

    const handleChange = (e) => {
        setNewTransaction({
            ...newTransaction,
            [e.target.name]: e.target.value,
        });
    };

    const handleAddTransaction = async () => {
        try {
            await createTransaction(categoryId, newTransaction);
            const data = await getTransactions(categoryId);
            setTransactions(data);
            setNewTransaction({ amount: '', note: '', transactionDate:
'' });
        } catch (error) {
            console.error('Failed to add transaction', error);
        }
    };

    const handleDeleteTransaction = async (transactionId) => {
        try {
            await deleteTransaction(categoryId, transactionId);
            const data = await getTransactions(categoryId);
            setTransactions(data);
        } catch (error) {
            console.error('Failed to delete transaction', error);
        }
    };

    const handleEditTransaction = (transaction) => {
        setCurrentTransaction(transaction);
        setNewTransaction({
            amount: transaction.amount,
            note: transaction.note,
            transactionDate: new
Date(transaction.transactionDate).toISOString().split('T')[0]
        });
        setEditMode(true);
    };

    const handleUpdateTransaction = async () => {
        try {
            await updateTransaction(categoryId,
currentTransaction.transactionId, newTransaction);
            const data = await getTransactions(categoryId);
            setTransactions(data);
            setNewTransaction({ amount: '', note: '', transactionDate:
'' });
            setEditMode(false);
            setCurrentTransaction(null);
        } catch (error) {
            console.error('Failed to update transaction', error);
        }
    };

    return (
        <div>
            <h2>Transactions</h2>
            <ul>
                {transactions.map(transaction => (
                    <li key={transaction.transactionId}>
                        Amount: {transaction.amount}, Note:
```

```
{transaction.note}, Date: {new
Date(transaction.transactionDate).toLocaleDateString()}
                        <button onClick={() =>
handleEditTransaction(transaction)}>Edit</button>
                        <button onClick={() =>
handleDeleteTransaction(transaction.transactionId)}>Delete</button>
                    </li>
                ))}
            </ul>
            <h3>{editMode ? 'Edit Transaction' : 'Add Transaction'}
</h3>
            <input
                type="number"
                name="amount"
                value={newTransaction.amount}
                onChange={handleChange}
                placeholder="Amount"
                required
            />
            <input
                type="text"
                name="note"
                value={newTransaction.note}
                onChange={handleChange}
                placeholder="Note"
                required
            />
            <input
                type="date"
                name="transactionDate"
                value={newTransaction.transactionDate}
                onChange={handleChange}
                required
            />
            <button onClick={editMode ? handleUpdateTransaction :
handleAddTransaction}>
                {editMode ? 'Update Transaction' : 'Add Transaction'}
            </button>
        </div>
    );
};

export default CategoryTransactions;
```

2. **Transaction Service**: Define functions to handle API requests for transactions.

```
import axios from 'axios';

const API_URL = 'http://localhost:8080/api';

export const getTransactions = async (categoryId) => {
    const token = localStorage.getItem('token');
    if (!token) throw new Error('No token found');

    const response = await
axios.get(`${API_URL}/categories/${categoryId}/transactions`, {
        headers: {
            Authorization: `Bearer ${token}`
        }
    });
    return response.data;
};

export const createTransaction = async (categoryId, transaction) => {
    const token = localStorage.getItem('token');
    if (!token) throw new Error('No token found');

    const response = await
axios.post(`${API_URL}/categories/${categoryId}/transactions`, {
        amount: parseFloat(transaction.amount),
        note: transaction.note,
        transactionDate: new
Date(transaction.transactionDate).getTime()
    }, {
        headers: {
            'Content-Type': 'application/json',
            Authorization: `Bearer ${token}`
        }
    });
    return response.data;
};

export const deleteTransaction = async (categoryId, transactionId) =>
{
    const token = localStorage.getItem('token');
    if (!token) throw new Error('No token found');

    const response = await
axios.delete(`${API_URL}/categories/${categoryId}/transactions/${trans
actionId}`, {
        headers: {
            Authorization: `Bearer ${token}`
        }
    });
    return response.data;
};

export const updateTransaction = async (categoryId, transactionId,
transaction) => {
    const token = localStorage.getItem('token');
    if (!token) throw new Error('No token found');

    const response = await
axios.put(`${API_URL}/categories/${categoryId}/transactions/${transact
ionId}`, {
        amount: parseFloat(transaction.amount),
        note: transaction.note,
        transactionDate: new
Date(transaction.transactionDate).getTime()
    }, {
        headers: {
            'Content-Type': 'application/json',
            Authorization: `Bearer ${token}`
        }
    });
    return response.data;
};
```

By following these steps, the frontend of the Expense Tracker Project is set up to handle user authentication, manage categories, and transactions, and provide a responsive user interface. The next section will cover deploying the project and additional features you might want to consider for enhancing the application.

Conclusion

The Expense Tracker project is a comprehensive application designed to help users manage their expenses effectively. By following the steps outlined in this report, you have created a robust backend using Spring Boot and PostgreSQL and a dynamic frontend using React. Additionally, you have explored deployment strategies and considered further enhancements to make the application more useful and engaging for users.

This project not only addresses a common concern for students studying abroad but also serves as an excellent demonstration of full-stack development skills. By continuing to iterate and improve on this project, you can create a valuable tool that benefits many users while further honing your technical abilities.

Future Improvements

While the Expense Tracker project provides a solid foundation for managing expenses, there are several areas where the application can be further enhanced to provide additional value and a better user experience. Here are some future improvements that can be considered:

1. Enhanced Security

- **Two-Factor Authentication (2FA)**: Implement 2FA to provide an additional layer of security for user accounts. This can be achieved through email or SMS-based verification codes.
- **OAuth Integration**: Allow users to log in using third-party services like Google or Facebook to streamline the authentication process and enhance security.

2. Improved User Interface

- **Mobile App**: Develop a mobile version of the Expense Tracker for iOS and Android platforms to provide users with access to their expenses on the go.

- **Responsive Design**: Ensure the web application is fully responsive and offers a seamless experience across all devices, including smartphones, tablets, and desktops.
- **Dark Mode**: Implement a dark mode to enhance user experience, especially in low-light environments.

3. Localization and Internationalization

- **Multi-Language Support**: Add support for multiple languages to cater to a global audience.
- **Currency Conversion**: Allow users to track expenses in different currencies and provide automatic currency conversion based on current exchange rates.

4. AI and Machine Learning

- **Expense Prediction**: Use machine learning algorithms to predict future expenses based on historical data, helping users plan their budgets more effectively.
- **Personalized Advice**: Provide personalized financial advice and tips based on users' spending habits and financial goals.

5. Enhanced Budgeting Features

- **Recurring Budgets**: Allow users to set up recurring budgets (e.g., monthly or weekly) and track their adherence to these budgets over time.
- **Savings Goals**: Enable users to set and track savings goals, with visual progress indicators and motivational messages.

Errors Encountered During Development

As with any development project, several challenges and errors were encountered during the creation of the Expense Tracker. Here, we will detail these issues and the solutions implemented to overcome them.

1. Sequence `ET_USERS_SEQ` Not Found

Error Description: There was an issue with the sequence `ET_USERS_SEQ` used for inserting data into the `ET_USERS` table. Despite the existence of the `et_users` table, the sequence was not found.

Solution:

1. **Create the Sequence Manually**:

```
CREATE SEQUENCE et_users_seq START 1;
```

2. **Modify the Table**:

```
ALTER TABLE et_users ALTER COLUMN user_id SET DEFAULT
nextval('et_users_seq');
```

3. **Update Entity Class**: Ensure that the `User` entity is configured to use the sequence:

   ```
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
   "et_users_seq")
   @SequenceGenerator(name = "et_users_seq", sequenceName =
   "et_users_seq", allocationSize = 1)
   private Integer userId;
   ```

4. **Verify Repository Configuration**: Check the `UserRepositoryImpl` to ensure it correctly uses the sequence:

   ```
   private static final String SQL_CREATE = "INSERT INTO
   ET_USERS(USER_ID, FIRST_NAME, LAST_NAME, EMAIL, PASSWORD)
   VALUES(NEXTVAL('et_users_seq'), ?, ?, ?, ?)";
   ```

5. **Spring Boot Configuration**: Ensure proper database configuration in `application.properties`:

   ```
   spring.datasource.url=jdbc:postgresql://localhost:5432/your_database
   spring.datasource.username=your_username
   spring.datasource.password=your_password
   spring.datasource.driver-class-name=org.postgresql.Driver
   spring.jpa.hibernate.ddl-auto=update
   spring.jpa.show-sql=true
   spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreS
   QLDialect
   ```

2. Ambiguous Routes

Error Description: During the creation of controllers, there were conflicts with handling identical routes in different controllers.

Solution:

- Ensure unique route mappings or use more specific routes to avoid conflicts. For example:

  ```
  @RequestMapping("/api/categories")
  public class CategoryResource { ... }

  @RequestMapping("/api/transactions")
  public class TransactionResource { ... }
  ```

3. PostgreSQL Authentication Failure

Error Description: An authentication failure occurred when creating the PostgreSQL server, as the default user was not detected.

Solution:

1. **Verify Password**: Ensure the correct password is being used. If forgotten, reset it.
2. **Reset PostgreSQL Password**:

```
sudo -i -u postgres
psql
ALTER USER postgres PASSWORD 'new_password';
\q
exit
```

3. **Configure `pg_hba.conf`**: Ensure it allows password authentication:

```
local   all             postgres                                md5
```

4. **Restart PostgreSQL**:

```
sudo systemctl restart postgresql
```

4. CORS Configuration Issues

Error Description: CORS issues when attempting to connect the frontend to the backend.

Solution: Configure CORS in Spring Boot:

1. **Global CORS Configuration**:

```
@Configuration
public class GlobalCorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://localhost:3000");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        source.registerCorsConfiguration("/**", config);
        return new CorsFilter(source);
    }
}
```

2. **Web Security Configuration**:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable()
            .authorizeRequests().antMatchers("/**").permitAll();
    }
}
```

3. **Web MVC Configuration**:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET, POST, PUT, DELETE, OPTIONS")
                .allowedHeaders("*")
                .allowCredentials(true);
    }
}
```

5. React Routing Issues

Error Description: React Router was not redirecting correctly, causing navigation issues after login or registration.

Solution:

1. **Ensure Proper Route Handling**:

```
<Route path="/login" element={<Login />} />
<Route path="/register" element={<Register />} />
<Route path="/dashboard" element={<PrivateRoute><Dashboard />
</PrivateRoute>} />
<Route path="/categories/:categoryId" element={<PrivateRoute>
<CategoryTransactions /></PrivateRoute>} />
<Route path="/edit-category/:categoryId" element={<PrivateRoute>
<EditCategory /></PrivateRoute>} />
```

2. **Handle Redirects After Login/Register**:

```
const handleLogin = async (event) => {
    event.preventDefault();
    try {
        const data = await login(email, password);
        localStorage.setItem('token', data.token);
        navigate('/dashboard');
    } catch (error) {
        alert('Login failed: ' + error.message);
    }
};

const handleRegister = async (event) => {
    event.preventDefault();
    try {
        await register(firstName, lastName, email, password);
        alert('Registration successful');
        navigate('/login');
    } catch (error) {
        alert('Registration failed: ' + error.message);
    }
};
```

6. Handling Token Expiry

Error Description: Issues related to handling expired tokens, causing unauthorized access errors.

Solution:

1. **Implement Token Expiry Handling**: Modify `AuthFilter` to check token validity:

```
try {
    Claims claims =
Jwts.parser().setSigningKey(Constants.API_SECRET_KEY)
            .parseClaimsJws(token).getBody();
    httpRequest.setAttribute("userId",
Integer.parseInt(claims.get("userId").toString()));
} catch (Exception e) {
    httpResponse.sendError(HttpStatus.FORBIDDEN.value(),
"invalid/expired token");
    return;
}
```

2. **Frontend Handling**: Redirect to login on token expiry:

```
axios.interceptors.response.use(
    response => response,
    error => {
        if (error.response.status === 401) {
            localStorage.removeItem('token');
            window.location.href = '/login';
        }
        return Promise.reject(error);
    }
);
```

7. SQL Syntax Errors

Error Description: SQL syntax errors in the repository classes causing runtime exceptions.

Solution:

- **Ensure Correct SQL Syntax**: Carefully review and test SQL queries in repository classes. For example:

```
private static final String SQL_CREATE = "INSERT INTO
ET_USERS(USER_ID, FIRST_NAME, LAST_NAME, EMAIL, PASSWORD)
VALUES(NEXTVAL('et_users_seq'), ?, ?, ?, ?)";
```

- **Use Prepared Statements**: Always use prepared statements to avoid SQL injection and ensure proper handling of SQL syntax.

```
jdbcTemplate.update(connection -> {
    PreparedStatement ps = connection.prepareStatement(SQL_CREATE,
Statement.RETURN_GENERATED_KEYS);
    ps.setString(1, firstName);
    ps.setString(2, lastName);
    ps.setString(3, email);
    ps.setString(4, hashedPassword);
    return ps;
}, keyHolder);
```

8. Dependency Injection Issues

Error Description: Errors related to dependency injection in Spring Boot, such as `NullPointerException` due to uninitialized beans.

Solution:

1. **Use `@Autowired`**: Ensure all dependencies are correctly injected using `@Autowired` annotation.

```
@Autowired
private UserRepository userRepository;
```

2. **Check Configuration**: Ensure Spring Boot scans the packages containing the annotated classes. This can be done by adding `@ComponentScan` in the main application class if necessary.

```
@SpringBootApplication
@ComponentScan(basePackages = "com.pairlearning.expensetracker")
public class ExpenseTrackerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ExpenseTrackerApplication.class, args);
    }
}
```

9. Incorrect Entity Mapping

Error Description: Incorrect mapping of entities causing issues with data retrieval and persistence.

Solution:

1. **Ensure Proper Annotations**: Verify that all entity fields are correctly annotated.

```
@Entity
@Table(name = "ET_USERS")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"et_users_seq")
    @SequenceGenerator(name = "et_users_seq", sequenceName =
"et_users_seq", allocationSize = 1)
    private Integer userId;

    @Column(name = "FIRST_NAME", nullable = false)
    private String firstName;

    @Column(name = "
```

LAST_NAME", nullable = false) private String lastName;

```
    @Column(name = "EMAIL", nullable = false, unique = true)
    private String email;

    @Column(name = "PASSWORD", nullable = false)
    private String password;

    // Getters and setters
```

}

~~~
2. **Database Synchronization**:
Ensure the database schema is in sync with the JPA entity definitions. This
can be managed using the `spring.jpa.hibernate.ddl-auto` property.
```properties
spring.jpa.hibernate.ddl-auto=update
~~~

Through addressing these errors and implementing robust solutions, the development of the Expense Tracker was not only a learning experience but also an opportunity to enhance problem-solving skills. Each challenge provided a deeper understanding of the technologies used and contributed to the overall quality and reliability of the application.