Overview

Expense Tracker is a full-stack web application for managing personal expenses. The repository contains a Spring Boot backend API, a React/Vite frontend, a PostgreSQL database, Docker Compose orchestration, and Traefik reverse proxy routing.

The application supports user registration, login, JWT-based access tokens, refresh tokens, and CRUD-style expense management.

Tech Stack

Area	Technology
Frontend	React 19, TypeScript, Vite, React Router
Styling/build tooling	Tailwind CSS tooling, ESLint, TypeScript compiler
Backend	Java 25, Spring Boot 4, Spring Web MVC
Security	Spring Security, JWT via JJWT, BCrypt password hashing
Persistence	Spring Data JPA, Hibernate, PostgreSQL
Migrations	Flyway
Reverse proxy	Traefik
Static serving	Nginx for the production frontend image
Containers	Docker, Docker Compose
Load testing	Shell script plus external HTTP load tooling

Repository Layout

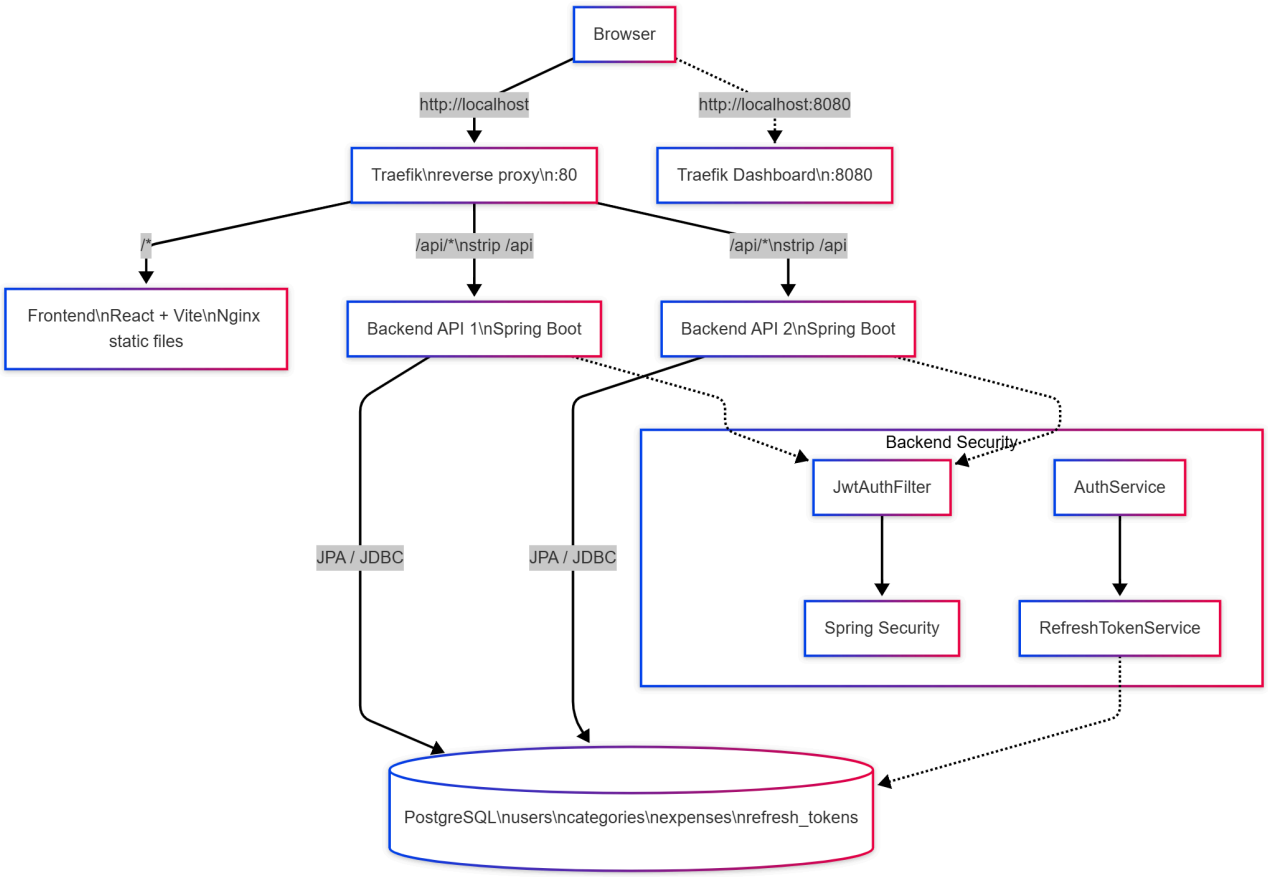
```
expensetracker/
  backend/                Spring Boot API
    src/main/java/...     Controllers, services, repositories, models
    src/main/resources/   application.yaml and Flyway migrations
    Dockerfile            Multi-stage backend production image
    pom.xml               Maven project and dependency config

  frontend/               React/Vite client
    src/                  App, auth screens, expense UI, API helpers
    Dockerfile            Multi-stage frontend production image
    nginx.conf            Nginx static file serving config
    package.json          Frontend dependencies and scripts

  docker-compose.yml      Base/prod-like stack
  docker-compose.dev.yml  Development overlay with hot reload
  README.md               Quick-start notes
  load-test.sh            API load-test helper
  document/               Generated or supporting documentation
```

Architecture

Mermaid Diagram



In production-like mode, Traefik listens on port 80 and routes `/api` traffic to the backend service. The frontend is served as static files by Nginx. The backend runs as stateless API containers and stores durable data in PostgreSQL.

In development mode, the same Traefik and database services are reused, but the backend runs through Maven with Spring Boot devtools and the frontend runs through the Vite dev server.

Request Flow

1. The browser loads the React app through Traefik.
2. Login or registration calls are sent to `/api/auth/...`.
3. Traefik strips the `/api` prefix before forwarding requests to the backend.
4. The backend returns an access token and refresh token after successful login.
5. The frontend stores tokens in local storage.
6. Protected expense API calls include `Authorization: Bearer <accessToken>`.
7. `JwtAuthFilter` validates the token and sets the authenticated user in Spring Security.
8. Controllers call services, services call repositories, and repositories use JPA/PostgreSQL.

Backend Architecture

The backend follows a common layered Spring Boot structure:

Layer	Responsibility	Examples
Controller	HTTP endpoints and request/response mapping	`AuthController`, `ExpenseController`, `HealthController`
DTO	API input/output data shapes	`LoginRequest`, `AuthResponse`, `CreateExpenseRequest`
Service	Business logic	`AuthService`, `ExpenseService`, `JwtService`, `RefreshTokenService`
Repository	Database access through Spring Data JPA	`UserRepository`, `ExpenseRepository`, `RefreshTokenRepository`
Model	JPA entities mapped to database tables	`User`, `Expense`, `Category`, `RefreshToken`
Config/filter	Security and request pipeline setup	`SecurityConfig`, `SecurityBeans`, `JwtAuthFilter`

Backend Endpoints

Endpoint	Method	Purpose	Auth required
`/health`	GET	Health check	No
`/auth/register`	POST	Register user	No
`/auth/login`	POST	Login and issue tokens	No
`/auth/refresh`	POST	Rotate refresh token and issue new access token	No
`/auth/logout`	POST	Revoke refresh token	No
`/expenses`	GET	List expenses with optional date filters and pagination	Yes
`/expenses`	POST	Create expense	Yes
`/expenses/{id}`	PUT	Update expense	Yes
`/expenses/{id}`	DELETE	Delete expense	Yes

Security Model

The backend uses stateless JWT security. `SecurityConfig` disables server-side sessions and requires authentication for all non-public routes. `JwtAuthFilter` checks the `Authorization` header, validates the bearer token, extracts the user identity, and places an authenticated principal into Spring Security's context.

Passwords are never stored directly. `SecurityBeans` provides a `BCryptPasswordEncoder`, which `AuthService` uses to hash passwords on registration and verify passwords on login.

Refresh tokens are persisted in the database as hashes. When refresh is called, `RefreshTokenService` rotates the token so the old token is replaced by a new one.

Database

Flyway manages the schema in `backend/src/main/resources/db/migration`.

Core tables:

Table	Purpose
`users`	Registered users, email, password hash
`categories`	User-specific expense categories
`expenses`	Expense records with amount, currency, date, note, and category

Table	Purpose
`refresh_tokens`	Hashed refresh tokens with expiration, revocation, and rotation metadata

Useful indexes include user email, expense user/date lookup, expense category lookup, and refresh token expiration.

Frontend Architecture

The frontend is a Vite React app. It separates reusable API/auth helpers from feature screens.

Important files:

File	Purpose
`src/main.tsx`	React app entry point
`src/App.tsx`	Main routing/application shell
`src/lib/api.ts`	Shared JSON POST helper and API error type
`src/lib/auth.ts`	Local token storage helpers
`src/features/auth/authApi.ts`	Login/register API calls
`src/features/auth/LoginPage.tsx`	Authentication UI
`src/features/expenses/expensesApi.ts`	Expense API client
`src/features/expenses/ExpensesPage.tsx`	Expense management UI

Docker And Runtime Modes

Production-like stack

```
docker compose --profile prod up -d --build
```

This uses `docker-compose.yml`. It builds production backend and frontend images, runs PostgreSQL, exposes Traefik on port 80, and exposes the Traefik dashboard/API on port 8080.

Services:

Service	Role
`traefik`	Reverse proxy and routing
`db`	PostgreSQL database
`backend-1`, `backend-2`	Backend API replicas
`frontend`	Nginx-served built React app

Development stack

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml --profile dev up --build
```

This layers `docker-compose.dev.yml` on top of the base compose file. It keeps shared services such as Traefik and PostgreSQL, but replaces the application runtime with hot-reload-friendly services.

Services:

Service	Role
`backend-dev`	Maven container running `spring-boot:run`
`frontend-dev`	Node container running Vite
`db`	Shared PostgreSQL service from the base compose file
`traefik`	Shared reverse proxy from the base compose file

Docker Images

The backend Dockerfile is a multi-stage build:

1. Maven image resolves dependencies and packages the Spring Boot application.
2. Eclipse Temurin JRE image runs the generated jar.

The frontend Dockerfile is also multi-stage:

1. Node image installs dependencies and runs the Vite production build.
2. Nginx image serves the generated `dist` files.

Environment Variables

The repository expects a local `.env` file based on `.env.example`. Important values include:

Variable	Purpose
`POSTGRES_DB`	Database name
`POSTGRES_USER`	Database user
`POSTGRES_PASSWORD`	Database password
`APP_JWT_SECRET`	Base64-encoded JWT signing secret
`APP_JWT_ISSUER`	JWT issuer value
`APP_JWT_ACCESS_EXPIRES_MINUTES`	Access token lifetime
`APP_JWT_REFRESH_EXPIRES_DAYS`	Refresh token lifetime

Operational Notes

- Application URL: `http://localhost`
- Traefik dashboard/API: `http://localhost:8080`
- Public auth routes are mounted under `/api/auth/...` through Traefik.
- The backend itself listens on port 8080 inside the container.
- The frontend production container serves static files on port 80 inside the container.
- JPA schema generation is set to `validate`; Flyway owns schema changes.

Load Testing

The repository includes `load-test.sh` for exercising the JWT-protected expense API with `hey`. The script logs in through `/api/auth/login`, registers the test user if needed, and sends authenticated requests to `/api/expenses`.

Supporting documentation is available in `document/load-test.md`. It records the measured GET results, including a 30-second run at 50 concurrent workers that handled about 12,356 requests per second with no failed requests. It also notes that an earlier POST check used an invalid payload and should not be treated as a write-throughput benchmark.

Summary

Expense Tracker is organized as a conventional, containerized full-stack app. The frontend handles UI and token storage, the backend owns authentication and expense business logic, PostgreSQL stores durable data, Flyway manages database changes, and Traefik provides a single local entry point for browser and API traffic.

The architecture is intentionally small but production-shaped: stateless backend services, explicit migrations, hashed passwords, refresh token rotation, reverse-proxy routing, and separate development and production container workflows.