Holiday Expense Tracker

Distributed Systems FS 2026

Holiday Expense Tracker

A fault-tolerant trip expense manager

Features

1. Supports login for multiple users.
4. Visual Representation of Expenses per Category.
5. Supports Speech-To-Text and Text-To-Speech.
6. Integrated AI Summaries and Money Saving Tips.
7. load balancing induced redundancy of the API component.
8. persistent storage of users, trips and expenses.
9. and many more ...

Short Facts

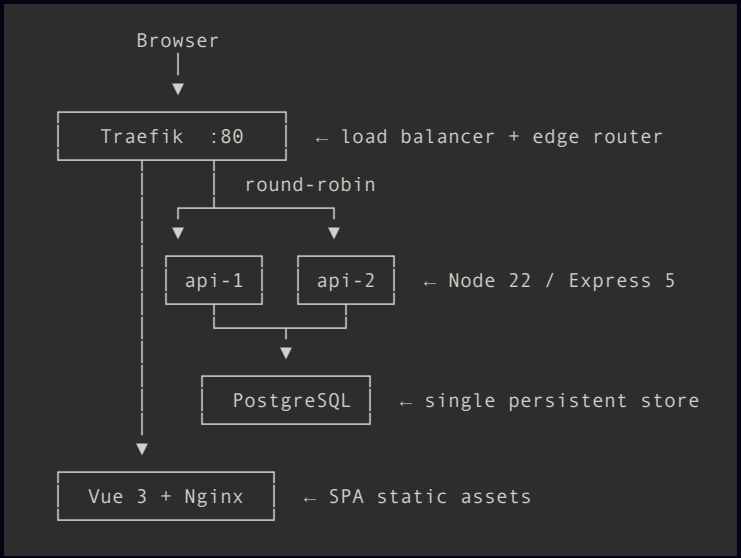
1. Built from scratch for the FS 2026 distributed systems challenge.
2. Assisted by Claude Code (Pro Subscription).
3. Time spent approximately 10h.

What was built

A multi-user web app for tracking travel expenses across trips.

- Login and register with JWT authentication
- Create trips, log expenses, filter by category
- Client-side currency conversion (USD / EUR / CHF)
- Runs across two redundant API instances

Architecture



▒▒ Traefik — Load Balancer

Routes all traffic and watches instance health.

- Polls `GET /health` on every container every 10 seconds
- Dead instance is removed automatically — no manual intervention
- Dashboard at `localhost:8080` shows live backend status

```
PathPrefix('/api') || PathPrefix('/health')  →  api-1, api-2
PathPrefix('/')                              →  Nginx (frontend)
```

▒▒ JWT — Why Failover Is Seamless

Tokens are self-contained meaning no shared session store is needed.

```
Login    →  server signs { sub, email, exp }  →  returns token
Request  →  any instance verifies with shared JWT_SECRET  →  done
```

- api-1 and api-2 share the same `JWT_SECRET` env var
- No database lookup on every authenticated request
- Kill api-1 mid-session → api-2 handles the next request without re-login

One-Command Setup

A fresh clone is fully running with a single command `make`

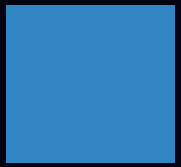
In the background the following happens:

1. npm installs dependencies.
2. npm creates the database seed (ensures proper bcrypt hashes).
3. docker compose up --build brings up the entire stack.

This leads to the following state:

1. DB schema auto-applied on first Postgres start
2. Seed: 2 users · 6 trips · ~94 expenses
3. Only manual step: copy `.env.example` → `.env`

```
# PostgreSQL password (used by both the postgres container and
backend DATABASE_URL)
POSTGRES_PASSWORD=changeme

# Generate one with: openssl rand -hex 32
JWT_SECRET=changeme

# OpenAI API key — required for the AI trip summary feature (task
05)
OPENAI_API_KEY=changeme
```



Demo

Running containers

To view the currently running containers run the command:

```
docker compose -p expense-tracker ps \
  --format "table {{.Name}}\t{{.Status}}"
```

snippet +exec is disabled, run with -x to enable

▒▒ Load balancing — live

Watch `X-Served-By` alternate between api-1 and api-2:

```
for i in 1 2 3 4 5 6; do
  curl -si localhost/health | grep X-Served-By
done
```

snippet +exec is disabled, run with -x to enable

Register a new user

We can now register a new user in the Database by sending the following curl request and see that the JWT token is returned.

```
curl -s -X POST localhost/api/auth/register \
  -H 'Content-Type: application/json' \
  -d '{"email":"tedy@example.com","password":"secret123"}' | jq
```

snippet +exec is disabled, run with -x to enable

Login and get a JWT

Alternatively we can also login with an existing user (added during seeding of database) and retrieve its JWT token.

```
curl -s -X POST localhost/api/auth/login \
  -H 'Content-Type: application/json' \
  -d '{"email":"alice@example.com","password":"password"}' | jq
```

snippet +exec is disabled, run with -x to enable

▒▒ Call a protected endpoint — no token

Now if we try to call one of the API endpoints without token we expect it to fail.

```
curl -s localhost/api/trips | jq
```

snippet +exec is disabled, run with -x to enable

▓▓ Call a protected endpoint — with token

Now if we retrieve the token of Alice and return it with jq for the curl command as $TOKEN we get a succesfull response.

```
TOKEN=$(curl -s -X POST localhost/api/auth/login \
  -H 'Content-Type: application/json' \
  -d '{"email":"alice@example.com","password":"password"}' | jq -r '
.token')

curl -s localhost/api/trips \
  -H "Authorization: Bearer $TOKEN" | jq '[.[] |
{id,name,destination}]'
```

snippet +exec is disabled, run with -x to enable

User isolation

Bob cannot read Alice's data even with a valid JWT.

```
BOB=$(curl -s -X POST localhost/api/auth/login \
  -H 'Content-Type: application/json' \
  -d '{"email":"bob@example.com","password":"password"}' | jq -r '
.token')

curl -s localhost/api/trips/1 \
  -H "Authorization: Bearer $BOB" | jq
```

snippet +exec is disabled, run with -x to enable

▓▓ Failover — kill api-2

Now to demonstrate the failover capabilities we kill the api-2 instance.

```
docker stop expense-tracker-api-2-1
sleep 3
echo "=== api-2 is down — all traffic on api-1 ==="
for i in 1 2 3 4; do
  curl -si localhost/health | grep X-Served-By
done
```

snippet +exec is disabled, run with -x to enable

Failover — recover api-2

```
docker start expense-tracker-api-2-1
sleep 12
echo "=== api-2 recovered ==="
for i in 1 2 3 4 5 6; do
  curl -si localhost/health | grep X-Served-By
done
```

snippet +exec is disabled, run with -x to enable

Load test

50 virtual users · JWT-protected endpoints

```
docker run --rm -i grafana/k6 run \
    -e BASE_URL=http://host.docker.internal \
    - < load-test/script.js
```

snippet +exec is disabled, run with -x to enable

```
  TOTAL RESULTS

    checks_total.......: 6001    117.660736/s
    checks_succeeded...: 100.00% 6001 out of 6001
    checks_failed......: 0.00%   0 out of 6001

    ✓ login → 200
    ✓ expenses → 200
    ✓ expenses → served-by
    ✓ trips → 200
```

Load test (killed API instance)

50 virtual users · JWT-protected endpoints

We run the load test in a docker container, followed by killing one of the redundant API instances:

```
docker run --rm -i grafana/k6 run \
    -e BASE_URL=http://host.docker.internal \
    - < load-test/script.js

docker stop expense-tracker-api-1
```

And receive the following (or similar) results, showing a mostly successful failover:

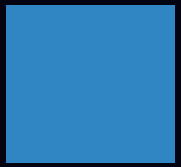
```
  TOTAL RESULTS

  checks_total.......: 5230   74.862368/s
  checks_succeeded...: 94.76% 4956 out of 5230
  checks_failed......: 5.23%  274 out of 5230

  ✓ login → 200
  ✗ expenses → 200
    ↳  98% — ✓ 1714 / ✗ 29
  ✗ expenses → served-by
    ↳  86% — ✓ 1516 / ✗ 227
  ✗ trips → 200
    ↳  98% — ✓ 1725 / ✗ 18
```

▓▓ Bottleneck: PostgreSQL

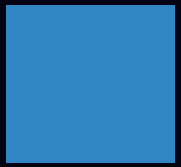
The API is stateless — add replicas at any time. The DB is the limit.

- 20 total DB connections under full load
- Every request runs a query — no cache
- p99 >> p50 means requests queue at the connection pool

To scale: PgBouncer (transaction pooling) or a read replica



Q&A



Thank you

▒▒ Make Commands

To run the project here a few helpful make commands. Have fun :)

```
make             →  full system up
make clean       →  wipe and rebuild
make load-test   →  k6 via Docker

localhost        →  frontend
localhost/api    →  backend (via Traefik)
localhost:8080   →  Traefik dashboard
```