Project information

Created on
February 24, 2026

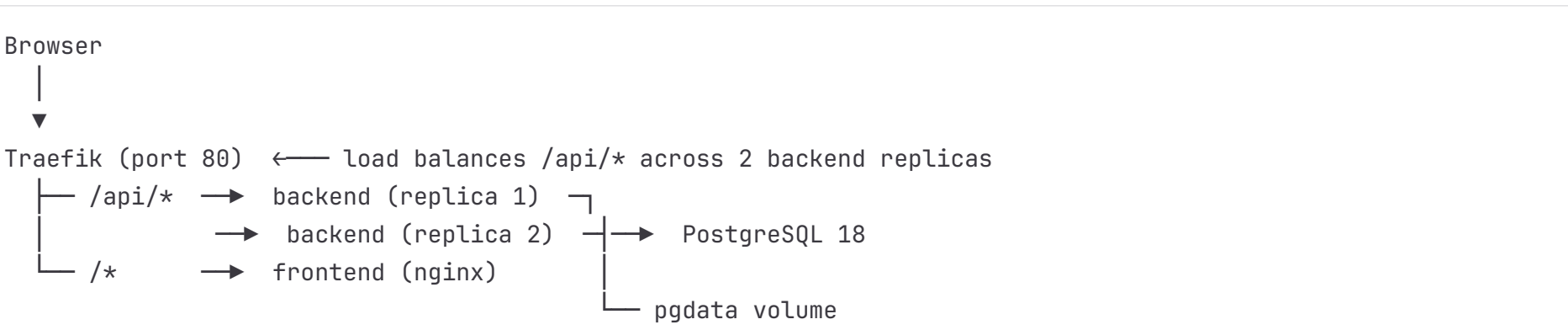
Name	Last update
plan	2 months ago
planli.backend	5 hours ago
planli.frontend	4 hours ago
.dockerignore	1 month ago
.gitignore	1 hour ago
README.md	3 hours ago
docker-compose.dev.yml	1 day ago
docker-compose.yml	1 day ago

README.md

Planli — Distributed Task Manager

A distributed task management system built with **ASP.NET Core 10**, **Blazor WebAssembly**, **PostgreSQL 18**, and **Docker**. Traefik acts as reverse proxy and load balancer across two backend replicas.

Architecture



Service	Technology	Notes
Frontend	Blazor WASM (.NET 10) served by nginx	Static files, no server-side rendering
Backend	ASP.NET Core 10 Web API	2 replicas, JWT auth, EF Core + Npgsql
Database	PostgreSQL 18	Persistent volume, health-checked
Proxy	Traefik 3	Path-based routing, round-robin load balancing

Prerequisites

- Docker Desktop (with Compose v2)

- `.env` file in the project root (see below)

Quick Start

1. Create the `.env` file

Create a file named `.env` in the project root (next to `docker-compose.yml`):

```
POSTGRES_USER=planli
POSTGRES_PASSWORD=planli_secret
POSTGRES_DB=planlidb
JWT_SECRET_KEY=super-secret-key-change-me-in-production-min-32-chars
JWT_ISSUER=planli-backend
JWT_AUDIENCE=planli-frontend
```

> **Production**: change `POSTGRES_PASSWORD` and `JWT_SECRET_KEY` to strong random values before deploying.

2. Build and start

```
docker compose up --build
```

3. Open the app

URL	Description
`http://localhost`	Planli frontend
`http://localhost/api`	REST API
`http://localhost:8081`	Traefik dashboard

Database migrations are applied automatically on backend startup.

4. Stop

```
docker compose down
```

To also wipe the database volume:

```
docker compose down -v
```

Development Setup (Visual Studio hot reload)

For local development, run only the database and pgAdmin in Docker and start the backend/frontend directly in Visual Studio.

1. Start the database

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

This starts:

- PostgreSQL on `localhost:5433`
- pgAdmin on `http://localhost:5050` (login: `admin@example.com` / `admin`)

2. Configure the backend

`planli.backend/planli.backend/appsettings.Development.json`:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Host=localhost;Port=5433;Database=planlidb;Username=planli;Password=planli_secret"
  },
  "Jwt": {
```

```
    "Secret":   "super-secret-key-change-me-in-production-min-32-chars",
    "Issuer":   "planli-backend",
    "Audience": "planli-frontend"
  }
}
```

3. Configure the frontend

`planli.frontend/planli.frontend/wwwroot/appsettings.Development.json`:

```
{
  "ApiBaseUrl": "http://localhost:5143"
}
```

4. Run in Visual Studio

Open `planli.backend.slnx` and `planli.frontend.slnx`, set each to the Development profile, and start both with hot reload (`Ctrl+F5` or the run button). Changes to `.razor` and `.cs` files reflect immediately without a full rebuild.

Authentication

The API uses **JWT Bearer tokens** with rotating refresh tokens, stored in **HttpOnly cookies** (OWASP recommendation — JavaScript cannot read them).

Token	Lifetime	Cookie path
Access token (`planli_access`)	15 minutes	`/` (all endpoints)
Refresh token (`planli_refresh`)	30 days	`/api/Auth` only

On a 401 response the frontend silently calls `/api/Auth/refresh` (the browser attaches the refresh cookie automatically). If refresh succeeds, new cookies are issued and the original request is retried. If refresh fails, the user is logged out.

Endpoints

```
POST /api/Auth/register   - create account, sets auth cookies
POST /api/Auth/login      - validates credentials, sets auth cookies
GET  /api/Auth/me         - returns current user info (reads access cookie)
POST /api/Auth/refresh    - rotates both tokens, sets new cookies
POST /api/Auth/logout     - invalidates refresh token in DB, clears cookies
```

API Overview

All endpoints except `/api/Auth/*` require `Authorization: Bearer <token>`.

Method	Path	Description
GET	`/api/Board`	List boards for current user
POST	`/api/Board`	Create board
PUT	`/api/Board/{id}`	Rename board
DELETE	`/api/Board/{id}`	Delete board
POST	`/api/Board/{id}/members`	Add member
DELETE	`/api/Board/{id}/members/{userId}`	Remove member
GET	`/api/ToDo?boardId={id}`	List tasks for board
GET	`/api/ToDo/{id}`	Get task detail
POST	`/api/ToDo`	Create task

Method	Path	Description
PUT	`/api/ToDo/{id}`	Update task
DELETE	`/api/ToDo/{id}`	Delete task

Board roles

Value	Role	Permissions
0	Owner	Full access, rename, delete board, manage all members
1	Editor	Create/edit/delete tasks, add viewers
2	Viewer	Read-only

Task priorities

Value	Label	Behaviour
0	Any Time	Lands in Backlog on creation
1	This Month	Lands in Backlog on creation
2	This Week	Lands in Backlog on creation
3	Today	Skips Backlog, goes straight to Not Started

Task statuses

`Backlog (4)` → `Not Started (0)` → `In Progress (1)` → `Completed (2)` / `Cancelled (3)`

Real-time Collaboration (Polling)

Changes made by one user appear for other users automatically via polling — no WebSockets required (intentional, to keep the load balancer simple).

Page	Poll interval
Board detail	5 seconds
Dashboard	15 seconds

Polling is paused while a modal is open or a drag-and-drop is in progress to avoid overwriting in-flight state.

Load Test Results

Tested against the full Docker stack (`docker compose up --build`) on a local development machine using hey. The tested endpoint was `GET /api/Board` — a JWT-protected endpoint that authenticates via cookie and queries the database.

All results

Test	Concurrency	Total requests	Duration	Req/sec	p50	p95	p99	Errors
Baseline	10	200	0.96s	208	14ms	329ms	660ms	0
Medium burst	50	1,000	1.55s	644	41ms	241ms	588ms	0
Heavy burst	100	2,000	2.23s	897	73ms	233ms	635ms	0
Sustained (c=100)	100	15,081	30s	496	44ms	999ms	1,289ms	**90 × 500**
Large run (c=50)	50	20,000	17.3s	**1,159**	38ms	85ms	130ms	0
Stress run (c=50)	50	100,000	82.2s	**1,216**	37ms	80ms	111ms	0

Key finding — concurrency sweet spot

The system performs best at **c=50 (50 concurrent workers)**:

- **100,000 requests completed with zero errors** at a sustained 1,216 req/sec
- p99 stays at 111ms — extremely consistent over 82 seconds
- At c=100 the connection pool is exhausted: throughput drops to ~496 req/sec and 0.6% of requests fail with HTTP 500

Sustained histogram — c=100 (pool exhaustion visible)

```
  0–181ms   ████████████████████████████████████  12,135 req  (fast path)
181–535ms   |                                        280 req
535–890ms   ██████                                 1,397 req  ← pool pressure
890ms–1.3s  ████                                   1,082 req
1.3s–1.8s                                            186 req
```

Stress histogram — c=50, 100k requests (stable)

```
  3–45ms    ████████████████████████████████████  65,501 req
 45–86ms    ████████████████                      30,949 req
 86–128ms   ██                                     3,117 req
128ms+                                               433 req
```

Analysis

`resp wait` (server processing time) dominates in every test. The bottleneck is **EF Core's database connection pool** — default maximum is 20 connections per process. With 2 backend replicas that gives 40 total connections.

- At **c=50**: requests are served smoothly within the available pool. Throughput peaks at ~1,216 req/sec with sub-millisecond variance.
- At **c=100**: 100 concurrent workers compete for 40 pool slots. Excess requests queue, p90 jumps from 70ms to 838ms, and some requests time out entirely (HTTP 500).

How to address it

Fix	Impact
`Maximum Pool Size=50` in the connection string	Doubles available connections per replica immediately
Add a third backend replica (`replicas: 3`)	Already supported by Traefik config — adds another 20 pool slots
Add a read replica for `GET` queries	Removes read load from the primary entirely