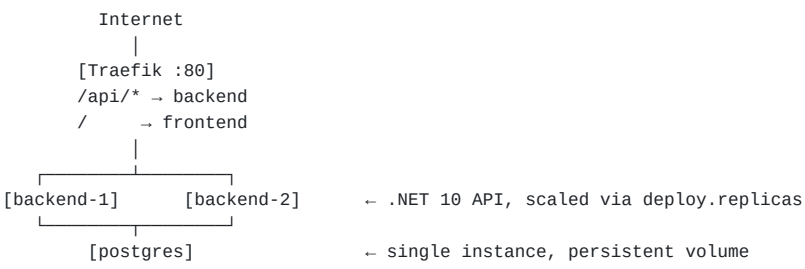
A distributed sports prediction game where you and your friends compete to call the correct match results.

CalledIt

A distributed football prediction game. Users register, place bets on match results, and compete on group and global leaderboards. Built as the FS 2026 Challenge Task — a horizontally-scaled, JWT-protected, fully containerized full-stack system.

Architecture



- **Backend layers**: `CalledIt.Api` (controllers, JWT) → `CalledIt.Domain` (services, validators) → `CalledIt.DataAccess` (EF Core, migrations).
- **Failover**: stopping `calledit-backend-2` mid-traffic does not produce client-visible errors — Traefik drops the dead replica from its pool and serves the remaining instance.
- **Stateless backend**: JWT authentication and an in-memory revocation cache hydrated at startup mean any replica can serve any request without sticky sessions.

Tech Stack

Layer	Technology
Backend	.NET 10 / ASP.NET Core Web API
Frontend	React 19, TypeScript, Vite, Chakra UI v3
Database	PostgreSQL 18 (alpine) via EF Core 10 + Npgsql
Auth	JWT Bearer (HS256) + `HttpOnly` fingerprint and refresh cookies (rotating)
Load balancer	Traefik v3.6
Containerization	Docker / Docker Compose v2
Load testing	wrk

Pinned versions live in `*.csproj`, `package.json` / `package-lock.json`, and the `Dockerfile`s — the manifests are the source of truth.

Authentication & Security

JWT setup follows the OWASP JWT Cheat Sheet. Login issues three artifacts:

1. **Access JWT** in the response body (HS256, signed with a 64+ char key, 15 min lifetime, claims: `sub`, `unique_name`, `jti`, `role`, `fgp`, exp). Stored client-side in `sessionStorage`.
2. **Fingerprint cookie** — random 32 bytes, `HttpOnly`, `SameSite=Strict`, `__Host-Fgp` in production, `Path=/`. The JWT carries `SHA256(fingerprint)` as the `fgp` claim; on every request the bearer pipeline recomputes the hash and compares with `CryptographicOperations.FixedTimeEquals`. This binds the token to the browser that requested it — a stolen JWT alone is useless.
3. **Refresh cookie** — 32 random bytes, base64url, `HttpOnly`, `SameSite=Strict`, `__Secure-Refresh` in production, `Path=/api/user/refresh`, 14 day lifetime. The browser only attaches it to the rotation endpoint, never to ordinary API calls.

Refresh token flow

Step	What happens
Login	Server issues access JWT + fingerprint cookie + refresh cookie. DB row in `RefreshTokens` stores `SHA256(rawToken)` — never the raw value.
API call → 401 (JWT expired)	Frontend `apiFetch` interceptor calls `POST /api/user/refresh` once (deduped via a shared in-flight promise so concurrent 401s don't burn multiple tokens), rewrites the `Authorization` header, and replays the original request.
Rotation on `/refresh`	Old row marked `RevokedAtUtc` + `ReplacedByTokenId` → new row, new JWT, new fingerprint, new refresh cookie. Strict one-time-use.
Bootstrap (tab reopened, `sessionStorage` empty)	Frontend tries `/api/user/refresh` once before showing the login screen — sessions survive tab close for 14 days.
Replay detection	If a refresh token that is *already revoked AND has a successor* is presented, every non-revoked refresh token for that user is wiped (`RevokeAllForUserAsync`).
Logout	Revokes the JWT's `jti` (blocklist) and calls `RevokeAllForUserAsync(userId)` for refresh tokens — "log out everywhere" by design, since the refresh cookie's path scoping means it isn't sent to `/logout`.

`JwtOptions.ExpirationMinutes` = access token lifetime (default 15). `JwtOptions.RefreshTokenDays` = refresh token lifetime (default 14).

Additional safeguards:

- **Token revocation**: logout writes the `jti` to a singleton `ConcurrentDictionary` and to `RevokedTokens` (durable). `JwtBearerEvents.OnTokenValidated` consults the in-memory cache only — no DB hit on the hot path. The cache is rehydrated on startup, and `pg_notify` broadcasts new revocations cross-replica so both backends see the same blocklist within milliseconds.
- **Brute-force lockout**: 5 failed logins → 15 min lockout (`failedLoginAttempts` + `lockoutEndUtc`).

- **Validation**: issuer, audience, signing key, and lifetime are all validated; `ClockSkew = 1 min`.
- **Role-based authorization**: admin endpoints (match/competition/team CUD, pending results) require `[Authorize(Roles = "Admin")]`.
- **Required config**: `JWT_KEY` is mandatory, has no fallback in source, and startup throws if it is missing or under 64 chars. Issuer, audience and expiry have safe defaults in `appsettings.json`.
- **CORS**: restricted to `localhost:5173`, `localhost:3000`, `localhost`.

Running

A fresh clone needs **one** manual step — putting secrets in `.env` — and then a **single** `docker compose up` command brings up the entire working system. Migrations run automatically. Seed data is opt-in via a Compose profile (still one command).

```
cp .env.example .env       # set DB_PASSWORD and JWT_KEY (only manual step — secrets)
```

```
Frontend:           http://localhost
Swagger UI:         http://localhost/swagger      (only if ENABLE_SWAGGER=true)
Traefik dashboard:  http://127.0.0.1:8080         (only if TRAEFIK_DASHBOARD=true)
```

Compose profiles

Profile	Command	What you get
(default — none)	`docker compose up --build`	Traefik, 2 × backend, frontend, Postgres, migrations applied. Empty database — register your own users.
`seed`	`docker compose --profile seed up --build`	Everything above **plus** the `seeder` job, which inserts users (admin + N test players), competitions, teams, matches and bets. Use this for the demo.
`seed-data`	`docker compose --profile seed-data up --build`	Like `seed` but skips user creation (`SEED_USERS=false`) — use it to top up fixtures against an existing user table.

`SEED_USER_COUNT` controls how many test players the `seed` profile creates (default 50). For a heavier demo:

```
SEED_USER_COUNT=100 docker compose --profile seed up --build
```

Startup ordering

```
postgres (healthy)
    └→ migrator (runs migrations, exits 0)
         └→ backend × 2 replicas
              └→ seeder (only when --profile seed / seed-data)
frontend, traefik (independent)
```

The `migrator` service uses the same backend image with `MIGRATE_ONLY=true`; backends only start once it has exited successfully. Seeders depend on the same gate, so they always run against an up-to-date schema.

Demo credentials (when seeded)

Account	Username	Password
Admin	`admin`	`AdminPassword1`
Player	`Delia`, `Hana`, `Keene`, …	`TestPassword1`

Environment variables

```
DB_NAME=calleditdb
DB_USER=calledit
DB_PASSWORD=<strong_password>
JWT_KEY=<openssl rand -base64 48>          # 64+ chars, mandatory
JWT_ISSUER=CalledIt
JWT_AUDIENCE=CalledIt.Client
JWT_EXPIRATION_MINUTES=15                  # access token lifetime
JWT_REFRESH_TOKEN_DAYS=14                  # refresh token lifetime

# Optional, off by default
ENABLE_SWAGGER=false
TRAEFIK_DASHBOARD=false
```

Load Testing

The load test targets a JWT-protected endpoint — `GET /api/leaderboard/{competitionId}` — using **wrk**. The script (`load-test/run-wrk.sh`) authenticates a real seeded user via `POST /api/user/login` , captures the JWT plus the fingerprint cookie, and runs wrk with both attached as headers.

Result

Two backend replicas behind Traefik, 16 threads × 200 connections × 30 s, no think time:

```
Running 30s test @ http://localhost/api/leaderboard/1
  16 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    73.53ms   52.50ms 620.17ms   75.46%
    Req/Sec   175.32    100.52     2.02k    85.38%
  83802 requests in 30.10s, 125.96MB read
Requests/sec:   2784.50
Transfer/sec:      4.19MB
```

~2,785 req/s sustained, fully authenticated, zero failed requests.

Bottleneck

At this throughput the dominant cost on the leaderboard hot path is the **JWT authentication pipeline**, not Postgres or the output cache. `UseOutputCache` runs *after* `UseAuthentication` / `UseAuthorization` , so even a cache hit pays the full price of HMAC-SHA256 signature verification, claim parsing, the in-memory revocation lookup, and the SHA-256 fingerprint-cookie hash on every request — roughly 7–8 CPU-ms per request, applied across both backends.

Postgres and the connection pool are not contended at this load — doubling backends gives ~2.13× throughput, the linear scaling signature of CPU-bound work rather than I/O contention. The denormalized `UserCompetitionStats` table plus a `(CompetitionId, TotalPoints DESC)` index serve the top-100 + caller's-rank query in a single window scan.

Failover demo

```
docker stop calledit-backend-2
# wrk continues — no client-visible errors, throughput halves
docker start calledit-backend-2
# throughput recovers as Traefik returns the replica to its pool
```

`http_req_failed` stays at 0 % — Traefik drops the stopped replica from its load-balancer pool immediately and every in-flight request is served by the surviving instance.

Running the load test

```
cp load-test/.env.example load-test/.env   # fill BASE_URL, COMPETITION_ID, TEST_USERS
./load-test/run-wrk.sh                     # run from WSL on Windows; needs wrk + jq
```

On Windows, run from WSL2 with mirrored networking (`networkingMode=mirrored` in `~/.wslconfig` , then `wsl --shutdown`).

Local Development

```
# 1. Start only the database
docker compose up postgres -d

# 2. Provide the JWT key as a user-secret (one-time, per machine)
cd src/Backend/CalledIt.Api
dotnet user-secrets set "Jwt:Key" "$(openssl rand -base64 48)"

# 3. Backend
cd ../..
dotnet run --project src/Backend/CalledIt.Api
# API: http://localhost:5004 — Swagger auto-enabled in Development

# 4. Frontend
cd src/Frontend/calledIt
npm install
npm run dev
# App: http://localhost:5173
```

EF Core migrations

```
dotnet ef migrations add <Name> \
  --project src/Backend/CalledIt.DataAccess \
  --startup-project src/Backend/CalledIt.Api
```

Applied automatically on backend startup in Development and Docker environments.

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 1

lukaszlr

Languages

C# 54.9% TypeScript 41.0% JavaScript 1.9% Shell 1.0% Dockerfile 0.8% PowerShell 0.2% Other 0.2%

Suggested workflows

Based on your tech stack

Deno Configure
Test your Deno project

Datadog Synthetics Configure
Run Datadog Synthetic tests within your GitHub Actions workflow

.NET Configure
Build and test a .NET or ASP.NET Core project.

More workflows Dismiss suggestions