FamilyTreeManager

Kurze Projektzusammenfassung

Wie auch schon der Name vom Projekt ahnen lässt, soll das Projekt das Managen von Stammbäumen erlauben. Ziel ist es, dass man mit dieser Anwendung mit einem einfachen Frontend Stammbäume visualiseren kann und diese auch ergänzen kann.

Agent

- Agent System → docs/agent/

Docker Start

1. `.env.example` nach `.env` kopieren.
2. Werte in `.env` lokal setzen.
3. Stack starten:

```
docker compose -f deploy/docker-compose.yml up --build
```

Der erste Docker-Schritt enthält: - nginx auf `http://localhost:8080` - Frontend und API gemeinsam auf `http://localhost:8080` - api1 und api2 intern hinter Nginx - `migrator` für einmalige EF-Migration vor API-Start - `postgres` auf `localhost:5432`

Loadtest ausführen

Standardmässig wird der Loadtest-User und Daten für den Loadtest geseedet. Dies kann aber im Docker Compose Datei abgeschaltet werden: entferne dafür den Flag `Database__SeedLoadTestData: "true"` im `migrator` Service.

Befehl für die Ausführung

Im Hintergrund wird hey für den Loadtest verwendet. Auf Windows, kopiere die hey Binaries (.exe) unter `loadtest/loadtest_binaries` mit dem Namen `hey_win.exe`. Auf anderen Plattformen (wie z.B. Linux) wird es entsprechend benannt: `hey_linux`. Als Fallback wird hey Befehl standardmässig verwendet (falls über ein Package-Manager installiert).

Befehlsbeispiel:

```
dotnet run --project loadtest/FamilyTree.LoadTest.csproj -- --scenario graph --
```

Derzeit kann auf 2 Endpunkten Loadtest ausgeführt werden: - `scenario graph`: jener Endpunkt, der viele Joins und Kalkulation im Backend macht ("langsamer" Endpunkt) - `scenario trees`: jener Endpunkt, der die Trees des Benutzers holt ("schneller" Endpunkt)

Distributed Family Tree Manager – Agent

Planning

Summary

This plan supersedes v2 and locks the UI model to match your target layout:

1. Right side: `family-chart` tree canvas (visualization + node selection).
2. Left side: custom Angular sidebar for person/relationship CRUD (add, edit, delete, add parent/sibling/partner/child).
3. Top toolbar: chart controls (zoom, fit, connection mode, export image).
4. Backend remains .NET + PostgreSQL with self-made JWT authentication.
5. Deployment remains Docker Compose with Nginx load balancing across two API instances.
6. Load testing must use an allowed open-source CLI tool against JWT-protected endpoints.

Final Product Scope (MVP)

1. Login/registration required via the application's own JWT auth before using app.
2. Users can create and manage their own family trees.
3. Person CRUD via left sidebar.
4. Relationship CRUD via left sidebar actions.
5. Interactive tree rendering on right via `family-chart`.
6. Distributed failover demo: one API instance can be stopped during runtime.
7. Load test report for protected endpoint(s).

Out of scope for MVP: 1. Real-time collaborative editing. 2. GEDCOM import/export. 3. Multi-tenant admin console beyond owner isolation.

Locked Technical Decisions

1. Backend: `ASP.NET Core Web API` (.NET LTS) + `EF Core` + `Npgsql`.
2. Frontend: Angular (standalone components) + `family-chart`.
3. Auth: self-made JWT auth in the API; API issues and validates signed access tokens with configured issuer/audience/signing key.
4. Load balancer: Nginx upstream with 2 API containers.
5. DB: single PostgreSQL instance with persistent volume.
6. Load test: allowed open-source CLI tool such as `hey`, `wrk`, or `ab`.
7. Bootstrap: fresh clone + single command `docker compose up --build`.
8. UI editing source of truth: sidebar forms/actions (not family-chart cards).

Architecture and Data Flow

1. User opens frontend and logs in or registers through the application auth UI.
2. Frontend receives a short-lived access JWT from `/api/v1/auth/login` or `/api/v1/auth/register`; refresh/fingerprint values are handled via HttpOnly cookies.
3. Frontend calls API through Nginx with `Authorization: Bearer <token>`.
4. Nginx routes requests to `api1`/`api2`.
5. API validates JWT and resolves `OwnerId` from `sub`.
6. API enforces ownership in all reads/writes.
7. Frontend loads graph data and renders with `family-chart`.

8. User selects a node in chart → selected person state updates sidebar.
9. Sidebar CRUD action → API call → on success, refresh graph data and rerender chart.
10. If `api1` fails, Nginx forwards to `api2` and app remains available.

UI/UX Specification (Decision Complete)

Layout

1. Left fixed sidebar (desktop): person details + action buttons.
2. Right main area: `family-chart` canvas.
3. Top toolbar above chart: zoom in/out, fit, toggle connection editing mode, export image.
4. Mobile behavior: sidebar collapses into slide-over panel.

Sidebar Behavior

1. If no node selected: show “Select a person” state and global “Add root person” action.
2. If node selected: show editable fields and actions:
3. Edit person
4. Delete person
5. Add parent
6. Add sibling
7. Add partner
8. Add child
9. Destructive actions require confirmation dialog.
10. Validation errors show inline and do not mutate chart state until success.
11. On successful mutation, chart reloads and keeps focus on affected person when possible.

family-chart Integration Rules

1. Use `family-chart` for rendering, pan/zoom, node selection, optional built-in utilities (export/fit).
2. Disable/ignore built-in edit cards as primary edit path.
3. Maintain Angular adapter module that maps API graph DTO ↔ family-chart node/link model.
4. Selected node ID is held in Angular state service and drives sidebar.

Public APIs / Interfaces

REST Endpoints

1. `GET /api/v1/trees`
2. `POST /api/v1/trees`
3. `GET /api/v1/trees/{treeId}`
4. `PATCH /api/v1/trees/{treeId}`
5. `DELETE /api/v1/trees/{treeId}`
6. `GET /api/v1/trees/{treeId}/persons`
7. `POST /api/v1/trees/{treeId}/persons`
8. `PATCH /api/v1/persons/{personId}`

9. `DELETE /api/v1/persons/{personId}`
10. `POST /api/v1/trees/{treeId}/relationships`
11. `DELETE /api/v1/relationships/{relationshipId}`
12. `GET /api/v1/trees/{treeId}/graph`
13. `GET /api/v1/health/live`
14. `GET /api/v1/health/ready`

Graph DTO Contract

`GET /api/v1/trees/{treeId}/graph` response: 1. `persons[]`: `id`, `firstName`, `lastName`, `displayName`, `birthDate`, `deathDate`, `gender`, `photoUrl?` 2. `relationships[]`: `id`, `type` (`PARENT_CHILD` | `PARTNER`), `fromPersonId`, `toPersonId` 3. `meta`: `treeId`, `version`, `generatedAt`

Sidebar Command Mapping

1. Add parent/child/sibling/partner → one or two API calls (create person if new, then create relationship).
2. Edit person → `PATCH /api/v1/persons/{personId}`.
3. Delete person → `DELETE /api/v1/persons/{personId}` with backend cascade/guard policy.
4. Delete relationship → `DELETE /api/v1/relationships/{relationshipId}`.

Domain and Persistence Specification

1. `family_trees(id, owner_id, name, description, created_at, updated_at)`
2. `persons(id, tree_id, first_name, last_name, birth_date, death_date, sex, photo_url, created_at, updated_at)`
3. `relationships(id, tree_id, from_person_id, to_person_id, type, created_at)`

Constraints: 1. FK integrity for tree/person/relationship. 2. Unique relationship tuple `(tree_id, from_person_id, to_person_id, type)`. 3. Relationship type check constraint. 4. Ownership enforced through tree linkage, never trust client owner IDs.

Indexes: 1. `family_trees(owner_id)` 2. `persons(tree_id)` 3. `relationships(tree_id)` 4. `relationships(from_person_id)` 5. `relationships(to_person_id)`

Security Specification

1. All API endpoints except health are JWT-protected.
2. API provides `/api/v1/auth/register`, `/api/v1/auth/login`, `/api/v1/auth/refresh`, `/api/v1/auth/logout`, and `/api/v1/auth/me`.
3. Passwords are hashed by the backend; refresh tokens are stored server-side as hashes and rotated through HttpOnly cookies.
4. API authorization policy checks token validity and owner-scoped access.
5. Return `401` for missing/invalid token, `403` for unauthorized resource access.
6. Frontend and API are served from one public origin via Nginx.

Distributed/Infra Specification

1. Services in compose:
2. `nginx`
3. `api1`
4. `api2`
5. `postgres`
6. Nginx upstream points to both APIs and handles failover.
7. API containers are stateless and horizontally replicated.
8. PostgreSQL persists data via named volume.
9. API startup runs migrations automatically before readiness.

Load Testing Specification

1. Obtain a JWT before the load test run and pass it via `Authorization: Bearer <token>`.
2. Target protected endpoint scenarios through Nginx:
3. `GET /api/v1/trees`
4. `GET /api/v1/trees/{treeId}/graph` for the seeded load-test tree
5. Use an allowed open-source CLI load-test tool:
6. `hey`: preferred default because it is simple, supports headers, and reports latency percentiles.
7. `wrk`: acceptable for higher-throughput tests if header/token handling is documented.
8. `ab`: acceptable fallback for a small smoke or baseline test.
9. Capture Docker runtime stats separately during the run for `familytree-nginx`, `familytree-api1`, `familytree-api2`, and `familytree-postgres`.
10. KPIs:
11. Throughput (RPS)
12. p50/p95/p99 latency
13. HTTP error rate
14. CPU and memory saturation per runtime container
15. Network I/O, block I/O, and PIDs per runtime container

Deliverable: 1. Load-test report artifacts with CLI tool output, Docker stats, bottleneck analysis, and max sustainable RPS.

Test Cases and Scenarios

Backend

1. Unit: entity validation and relationship rules.
2. Integration: CRUD endpoints against PostgreSQL container.
3. Auth:
4. No token → `401`
5. Invalid token → `401`
6. Foreign tree access → `403`

Frontend

1. Login flow works and token attached to API calls.

2. Node selection updates sidebar content.
3. Sidebar “add child/partner/parent/sibling” updates chart after API success.
4. Delete flows require confirmation and reflect in chart.
5. Error handling: API failure shows toast and keeps UI consistent.

End-to-End CT Demo

1. Fresh clone + `docker compose up --build`.
2. Login and create/edit tree via sidebar.
3. Show requests balanced across both APIs.
4. Stop one API container during interaction.
5. Continue successful operations with remaining API.
6. Present load-test results, Docker stats, and explain bottleneck.

Implementation Work Packages (Execution Order)

1. Scaffold solution structure and projects.
2. Implement domain model + EF Core migrations.
3. Implement JWT auth and ownership authorization.
4. Implement tree/person/relationship REST API.
5. Implement graph endpoint and Angular adapter.
6. Build Angular shell layout (left sidebar + right chart + top toolbar).
7. Wire chart selection state to sidebar.
8. Implement sidebar CRUD actions and API orchestration.
9. Integrate Nginx + compose replication (`api1`, `api2`) + health checks.
10. Finalize self-made JWT auth bootstrap/configuration for local Docker demo.
11. Add test suites and CI commands.
12. Add allowed open-source load-test command(s), Docker stats capture, and reporting docs.
13. Prepare presentation runbook.

Assumptions and Defaults

1. `family-chart` supports required rendering and selection APIs in chosen stable version.
2. Sidebar is the only editing UI required for MVP.
3. Latest stable framework/library versions will be used at implementation time.
4. Single PostgreSQL instance satisfies CT requirements.
5. Deployment target is local Docker environment used in course presentation.
6. HTTPS termination is not mandatory for local CT demo unless supervisor requests it.