User Connections & Discovery

End-to-end full-stack feature · iQuirium learning & case-portfolio platform

Engineer: Erich Assuncao **Role:** Sole full-stack developer (data model -> delivery) **Delivered:** June 2026

OVERVIEW

Summary

Built the entire user-connections capability for iQuirium, a professional development platform where people also publish learning cases and portfolios - from the database up. Before this work, members had no way to connect with one another at all: no relationship model, no invitations, no social graph. I designed and implemented the connection data model and its lifecycle, the full set of invitation and relationship actions, and then a discovery surface on top so members can find and connect with each other directly. The work covered the complete lifecycle - data modelling, backend domain logic and APIs, frontend UI, automated tests at every layer, and local developer tooling - and is now running in production.

ROLE & SCOPE

What I owned

I delivered this capability independently, across every layer of the stack:

- **Data model & persistence -** designed the connection relationship entity, its state machine, and the relational schema; authored the Entity Framework Core migrations that introduced and later hardened it.
- **Backend domain & APIs -** the connection aggregate with enforced invariants, the full invitation/relationship lifecycle as CQRS commands and queries, validation, and OpenAPI docs.
- **Frontend -** the connections experience (connections, received, sent, and discover), relationship-aware action buttons, search, state management, internationalised copy, and a typed API client generated from the backend contract.
- **Quality -** ~220 automated tests across domain, application, infrastructure, API, and UI layers, plus a scripted manual smoke test for the full invite/accept flow.
- **Developer tooling -** local mock connection users and a sign-in helper so the multi-user experience can be exercised end-to-end on a developer machine.

CONTEXT

The problem

iQuirium had no concept of a connection between users. Members could publish public cases, but there was no relationship model in the database, no invitations, and no way for two people to link their accounts - so no social graph could exist. A connections capability had to be built from the ground up: the data model first, then the actions that operate on it, and finally a way for people to find each other.

WHAT I BUILT

The solution

1. A connection capability, built from the database up

A reversible, single-record relationship between two users, modelled as a proper domain aggregate with its rules enforced in code rather than scattered across handlers:

- **One canonical record per user pair.** Each relationship is stored once with a deterministic ordering of the two user IDs, so duplicate or mirror-image rows are impossible by construction.
- **A reversible state machine.** A connection moves through Pending -> Accepted / Declined -> Removed, with Blocked able to supersede any state, and declined or removed pairs able to be reinvited - every transition guarded so illegal moves throw rather than corrupt state.
- **Encapsulated invariants.** Only the recipient can accept or decline, only the sender can cancel, only the participant who blocked can unblock, and a user can never connect to themselves - all enforced inside the aggregate.
- **Asymmetric blocking with a cool-down.** Blocking records who initiated it and applies a 48-hour re-block cool-down to prevent block/unblock churn, mirroring mainstream social networks.
- **Persisted and migrated safely.** Backed by a relational schema and a sequence of EF Core migrations that introduced the entity and later hardened its lifecycle without breaking existing data.

2. The full invitation & relationship lifecycle

On top of the model, a complete set of actions and read APIs: send, accept, decline, and cancel invitations; remove a connection; block and unblock; and queries for a pair's relationship status, a user's connections, and their received and sent invitations - each as its own validated, tested CQRS command or query.

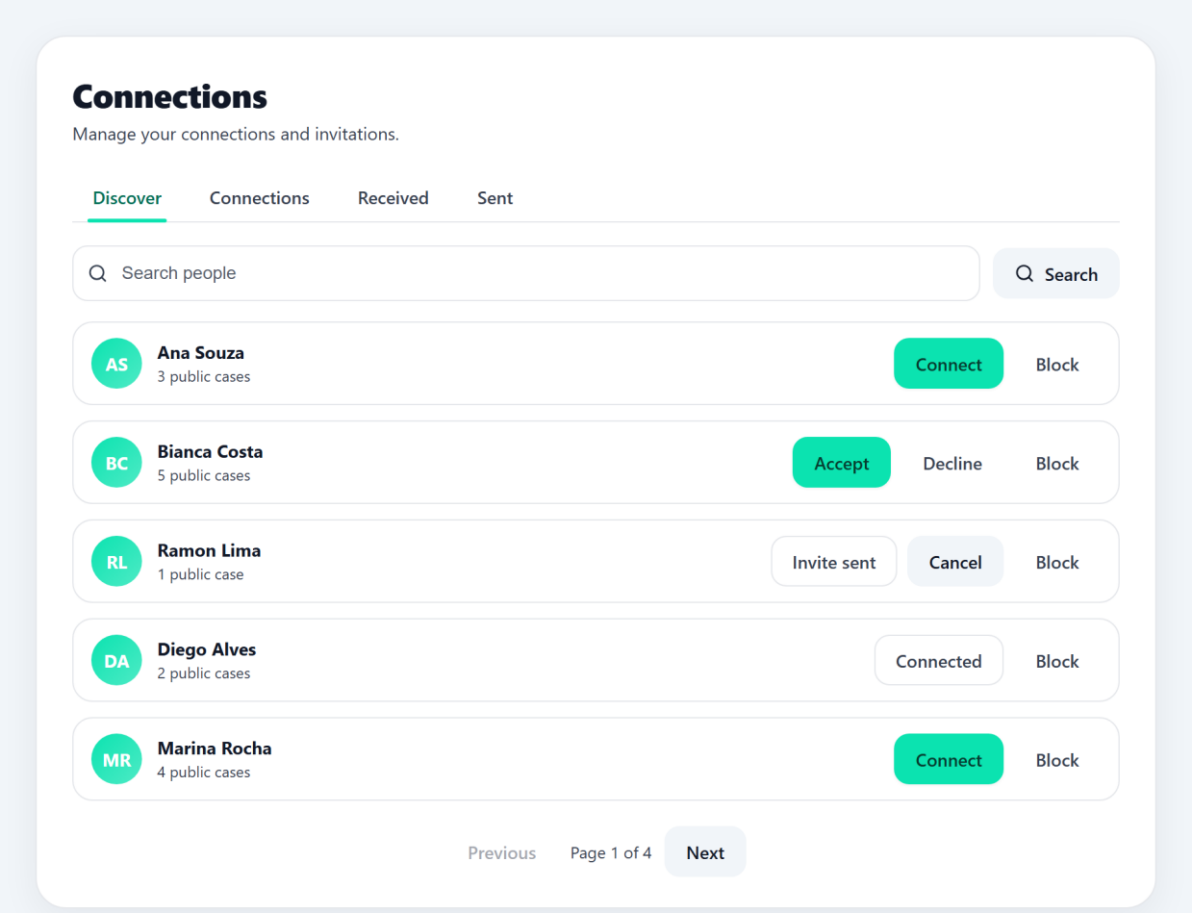
3. A discovery surface to activate the graph

Finally, a LinkedIn-style Discover surface on the Connections page so members can grow the graph without already knowing a profile link:

- **Discover, search & connect** from one screen - name search with debounced input, paginated relevance-ranked results, and direct links to public profiles.
- **Relationship-aware rows** that render the correct action per person (Connect, Invite sent / Cancel, Accept / Decline, Connected, Block) from the live relationship state.
- **Privacy by design** - only users with public content are discoverable, emails are never searchable or exposed, and blocked relationships are excluded from results.

INTERFACE

The Discover surface



The Discover tab: name search, relevance-ranked results, and a context-aware action per row (Connect / Accept-Decline / Invite sent-Cancel / Connected) driven by the live connection state. Representative rendering using local mock data.

ENGINEERING

Architecture & technical highlights

Domain & data

- **Rich domain model.** The relationship is a true aggregate - private setters, factory methods, and guard clauses - so the connection lifecycle can only ever be in a valid state, independent of which API path touches it.
- **Safe schema evolution.** Delivered as incremental EF Core migrations (introduce, then harden, then extend with block direction), keeping existing rows valid at every step.

Backend

- **CQRS throughout.** Each action and read is a dedicated command/query with its own validator and handler, following the codebase's established conventions; handlers stay thin and delegate state changes to the aggregate.
- **Single-query discovery.** The discovery endpoint returns each candidate with its full relationship snapshot and permitted actions, avoiding an N+1 pattern of one status call per row.
- **Shared relationship mapper.** Relationship/permission logic is centralised in one mapper reused by both discovery and the per-user status endpoint, so the two surfaces can never drift apart.
- **Contract-first.** Regenerated the OpenAPI specification so the frontend client and types come directly from the backend contract.

Frontend

- **Type-safe, generated client.** API hooks and models are generated from the OpenAPI spec, binding the UI to the real backend contract end-to-end.
- **Server-state management.** After any mutation the relevant caches (discovery, connections, sent, received) are invalidated so every tab stays consistent.
- **Composed, tested UI.** Reusable row, CTA, and action-hook components with explicit loading, empty, search-empty, and error/retry states - each covered by unit tests.

STACK

Technology

Domain / Data	Domain-driven aggregate design, relational schema, Entity Framework Core migrations
Backend	C# / .NET, CQRS, Entity Framework Core, FluentValidation, OpenAPI / Swagger
Frontend	TypeScript, Next.js, React, TanStack Query, Orval (typed client generation), next-intl
Testing	xUnit (domain, application, infrastructure, API), Vitest + Testing Library (UI)
Practices	PR-based delivery, contract-first API, test-first, privacy-by-design, local mock tooling

ENGINEERING RIGOR

Quality & delivery

- **~220 automated tests** across the domain entity, application handlers, infrastructure repositories, API endpoints, and frontend components - the connection aggregate alone is covered by 50+ unit tests exercising every state transition and guard.
- **Documented manual smoke test** for the complete two-user journey: discover -> search -> view profile -> invite -> accept -> confirm both sides connected.
- **Incremental, reviewable delivery.** Shipped as a sequence of focused, reviewed pull requests rather than one large change - relationship status, then invite, accept/decline, cancel/remove, block/unblock, and finally discovery, each with its own tests.
- **Verified end-to-end and deployed to production.** The full capability is deployed and running in production, demonstrating delivery from an empty data model to a live, integrated build.

TAKEAWAYS

Skills demonstrated

Domain-driven design & data modelling · state-machine design · database schema & safe migrations · API and domain design · CQRS and clean-architecture patterns · performance-conscious data access · React UI engineering and server-state management · contract-first full-stack integration · comprehensive multi-layer testing · privacy-aware feature design · independent end-to-end ownership.

Confidentiality - iQuirium is a private product. This document describes the feature, architecture patterns, and engineering approach at a high level and intentionally omits proprietary source code, business logic, credentials, and internal infrastructure details.