

Teil B

Dieses Dokument ergänzt die praktische Umsetzung von Teil A (Leave-Planner) um die begründete Architekturperspektive. Ich habe es so aufgebaut, dass Fach- und Technik-Stakeholder nachvollziehen können, warum die Lösung so gestaltet ist, welche Annahmen gelten, wo die Grenzen liegen und wie die Lösung schrittweise skaliert und weiterentwickelt werden kann.

Inhalt & Leitfragen:

- **Zielbild & Annahmen:** Zweck, Nutzergruppen, Kernereignisse, Systemgrenzen; explizite Annahmen und Randbedingungen für v1.
- **Architektur der Softwarelösung:** Schichten, Verantwortlichkeiten, Schnittstellen. Architekturentscheidungen (z. B. Ein-Tages-Antrag, projektbezogene Konflikte, serverseitige Guards) sowie der Evolutionspfad (Auth, Services, Outbox, Azure).
- **Persistenz & Datenmodell:** Kerntabellen und Constraints für Integrität/Idempotenz, Muster für Konsistenz & Parallelität, Optionen für Historisierung.
- **Skalierung & Performance:** Erwartete Lastmuster, mögliche Engpässe und Gegenmaßnahmen (Pagination, Caching, DB-Upgrade, Observability).

1) Zielbild & Annahmen

1.1 Zweck

Mitarbeitende sollen eintägige Urlaubsanträge stellen können. Entscheidungsträger*innen (Approver) sehen dabei Konflikthinweise pro Projektteam am gewählten Tag. So werden gleichzeitige Abwesenheiten in denselben Projekten früh sichtbar und Risiken für Lieferfähigkeit & Planung reduziert.

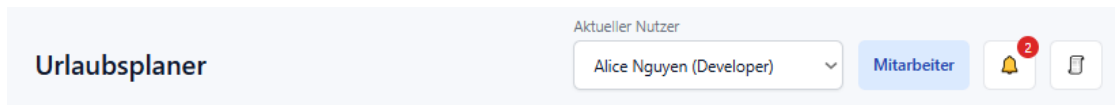
1.2 Ausgangslage & Nutzen (Business Value)

- **Ausgangslage:** Urlaube werden heute oft verteilt oder verspätet kommuniziert. Erst im Meilenstein fällt auf, dass „die halbe Crew“ weg ist.
- **Nutzen:** Das System zeigt sofort bei Antragstellung bzw. spätestens vor der Genehmigung, ob im gemeinsamen Projekt bereits Abwesenheiten liegen. Approver können begründet entscheiden, Anträge staffeln oder Alternativen schaffen (Vertretung, Prioritäten).

1.3 Rollen & Verantwortungen

Alle Nutzer

- **User-Auswahl im Header:** Setzt die Identität (GUID) für alle nachfolgenden Requests.



- **Company Overview:** Sieht alle Projekte der Firma inkl.
 - Kunde, Projektlaufzeit
 - Assignments
 - genehmigte Abwesenheiten im gewählten Monat

The 'Unternehmensübersicht' (Company Overview) page for September 2025. The page title is 'Unternehmensübersicht' with the subtitle 'Projektzuweisungen und genehmigte Abwesenheiten im Unternehmen.' (Project assignments and approved absences in the company). The page shows details for 'Project A1' (Customer A | 2025-01-01 bis 2025-12-31). Below the project details, there is a section for 'Teammitglieder (3)' (Team members (3)) listing 'Carlos Diaz', 'Alice Nguyen', and 'Bob Meier'. The main section is 'Genehmigte Abwesenheiten im September' (Approved absences in September), which displays a calendar grid. The calendar shows the days of the month, with the date '16' highlighted in yellow and labeled 'Alice', indicating her approved absence for that day.

Mo	Di	Mi	Do	Fr	Sa	So
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16 Alice	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Employee (Mitarbeitende:r)

Eintägige Abwesenheiten unkompliziert beantragen und sofort sehen, ob es am gewählten Tag in eigenen aktiven Projekten zu Engpässen kommen könnte.

- **Tab „Request Leave“:** Datum wählen, Antrag absenden. Direkt danach erscheint ein Konflikt-Hinweis (Toast), falls Kolleg:innen im selben aktiven Projekt bereits approved sind.
- Pro (EmployeeId, Date) nur 1 Antrag
- **Konfliktcheck bei Antragstellung** nur gegen Approved anderer Teammitglieder in gemeinsamen, aktiven Projekten.

Urlaub beantragen

Mitarbeiter: Alice Nguyen

Datum: mm/dd/yyyy

Hinweis: Beim Beantragen werden nur bereits **genehmigte** Abwesenheiten anderer im selben Projekt als Konflikt berücksichtigt.

Urlaub beantragen

Meine Anträge

DATUM	STATUS	KOMMENTAR	ENTSCIEDEN VON / AM
2025-09-16	Genehmigt	—	Sofia Supervisor • 9/18/2025, 8:56:26 PM

- **Notifications Drawer:** Posteingang für Ereignisse („submitted“, „approved“, „rejected“), Filter „nur ungelesene“, Button „alle sichtbaren markieren“. Der Header zeigt eine **Badge** mit Anzahl ungelesener Nachrichten.

Liste der neuesten Benachrichtigungen:

- **Icon/Typ:** *submitted, approved, rejected*
- **Datum** (der betroffene Urlaubstag) & **Zeitstempel** (Erzeugung)
- **Akteur** (wer entschieden hat) & **Kommentar** (falls vorhanden)
- **Statuspunkt:** ungelesen (markiert) oder gelesen

Primäraktionen:

- **Refresh** – lädt die Liste neu.
- **Schließen** – schiebt den Drawer zu.
- **Filter:** Checkbox „nur ungelesene“ (unter den Buttons platziert, um den Fokus auf die Aktionen zu halten).

- **Massenaktion:** Button „Alle sichtbaren markieren“ (markiert nur die aktuell gefilterten & angezeigten Items als gelesen).
- **Empty State:** Bei leerer Liste deutlicher Hinweis, z. B. „*Keine (ungelesenen) Benachrichtigungen*“.
- **Notifications lesen/markieren** erfordert gültige Identität → sonst error

Benachrichtigungen

Neu laden

Schließen

☐ nur ungelesene

2 Einträge

Alle sichtbaren als gelesen

☐ **Antrag genehmigt** 9/18/2025, 8:56:27 PM
Datum: 2025-09-16
von: Sofia Supervisor

☐ **Antrag eingereicht** 9/18/2025, 5:15:15 PM **neu**
Datum: 2025-09-16

Markierte als gelesen

Approver (z. B. Team-Lead/PO)

Entscheidungen treffen; vor Genehmigung die vollständige Konfliktlage im Team sehen (Approved und Requested anderer am gleichen Tag im gleichen Projekt).

- **Tab „Manage Leaves“:** Offene Anträge prüfen, Approve/Reject.
Sortierung: nach Datum (aufsteigend).
Aktionen je Zeile:
 - **Konflikte prüfen** (optional): zeigt projektbezogene Konflikte (Approved + Requested anderer Teammitglieder am selben Tag)
 - **Approve:** genehmigt den Antrag.
 - **Reject:** lehnt ab (Dialog mit optionalem Kommentar).

Bereits Approved/Rejected Anträge sind nicht entscheidbar; die API liefert in diesen Fällen 409 Conflict.

Urlaubsanträge verwalten

Hinweis: Konflikte berücksichtigen genehmigte & angefragte Abwesenheiten.

MITARBEITER	DATUM	STATUS	KONFLIKTE
Alice Nguyen	2025-09-16	Genehmigt	–
Fatima Ali	2025-09-13	Genehmigt	<div><div>⚠ Konflikte in zugehörigen Projekten</div><div>Project B1 1 genehmigt</div><div>Genehmigt: Eren Kaya</div></div>
Eren Kaya	2025-09-13	Genehmigt	<div><div>⚠ Konflikte in zugehörigen Projekten</div><div>Project B1 1 genehmigt</div><div>Genehmigt: Fatima Ali</div></div>
Alice Nguyen	2025-09-12	Angefragt	– ✓ ✗
Daria Novak	2025-09-11	Genehmigt	<div><div>⚠ Konflikte in zugehörigen Projekten</div><div>Project B1 1 genehmigt</div><div>Genehmigt: Eren Kaya</div></div>
Eren Kaya	2025-09-11	Genehmigt	<div><div>⚠ Konflikte in zugehörigen Projekten</div><div>Project B1 1 genehmigt</div><div>Genehmigt: Daria Novak</div></div>
Fatima Ali	2025-09-01	Abgelehnt	<div><div>⚠ Konflikte in zugehörigen Projekten</div><div>Project B1 1 angefragt</div><div>Angefragt: Daria Novak</div><div>Project B2 1 angefragt</div><div>Angefragt: Daria Novak</div></div>

- **Rollen-Guard:** Approve/Reject nur für Approver/Admin → sonst 403 Forbidden.
- **Self-Approve/-Reject verboten** → 403 Forbidden.
- **Audit-Felder** werden gesetzt: DecisionBy, DecisionAt (UTC), optional DecisionComment.
- Nach Entscheidung erzeugt der Server eine **Notification** für die/den Antragsteller:in.

Admin

Stammdaten & Rollen pflegen; Governance gewährleisten, ohne versehentlich die Admin-Funktionalität unbrauchbar zu machen.

- **Tab „Admin Panel“:**
 - **Employees:** anlegen, bearbeiten (Name, JobTitle, Role), löschen.



















Benutzerverwaltung

Projektverwaltung

Kundenverwaltung

Benutzer

Benutzer hinzufügen

NAME	ROLLE	JOBTITEL	
Amira Admin	Admin	Admin	 
Peter Product	Genehmiger	Approver	 
Sofia Supervisor	Genehmiger	Approver	 
Alice Nguyen	Mitarbeiter	Developer	 
Bob Meier	Mitarbeiter	Developer	 
Carlos Diaz	Mitarbeiter	Developer	 
Daria Novak	Mitarbeiter	Developer	 
Eren Kaya	Mitarbeiter	Developer	 
Fatima Ali	Mitarbeiter	Developer	 

- **Customers:** anlegen/bearbeiten/löschen.





Benutzerverwaltung

Projektverwaltung

Kundenverwaltung

Kunden

Kunde hinzufügen

NAME	
Alpha	 
Beta	 

- **Projects:** anlegen/bearbeiten/löschen (Name, Kunde, Start/Ende).

Benutzerverwaltung

Projektverwaltung

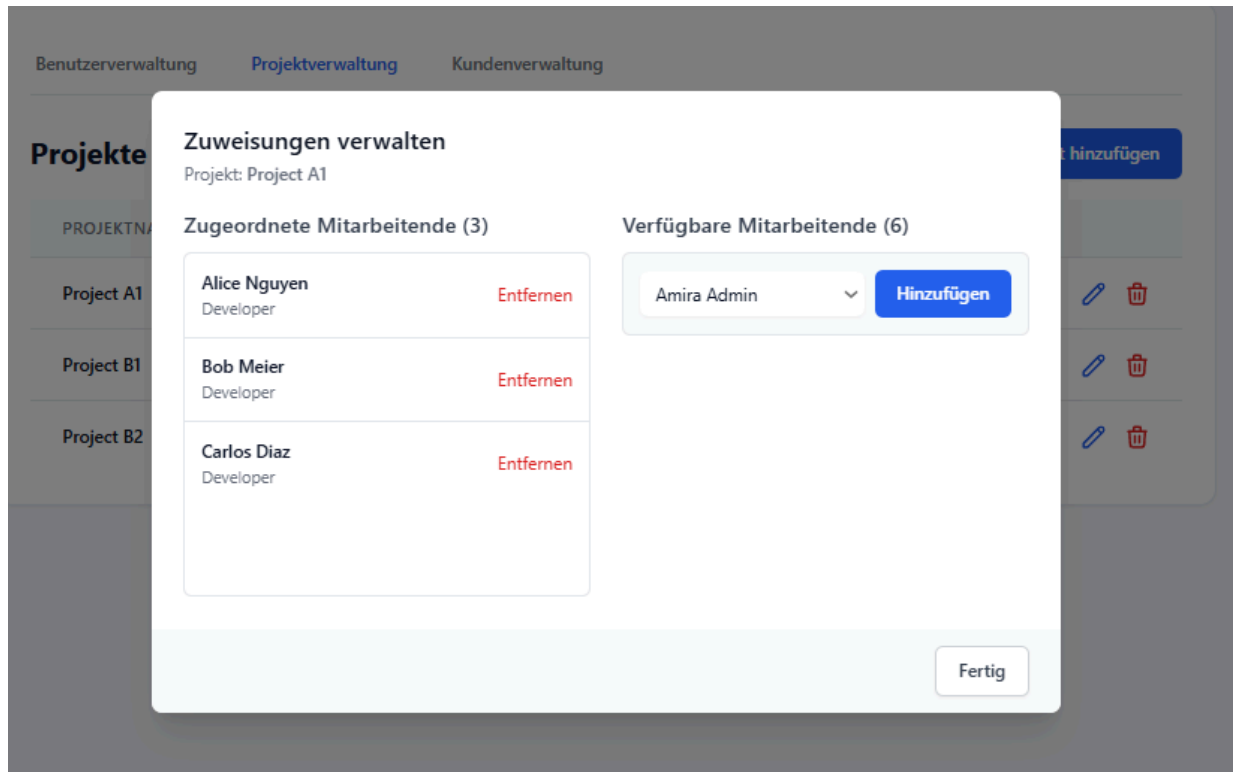
Kundenverwaltung

Projekte

Projekt hinzufügen

PROJEKTNAME	KUNDE	ZEITRAUM	ZUGEORDNET	
Project A1	Customer A	2025-01-01 - 2025-12-31	3	  
Project B1	Customer B	2025-03-01 - 2025-11-30	3	  
Project B2	Customer B	2025-09-01 - 2026-03-31	3	  

- **Assignments:** Mitarbeitende Projekten zuordnen/entfernen (Duplikate verhindert).



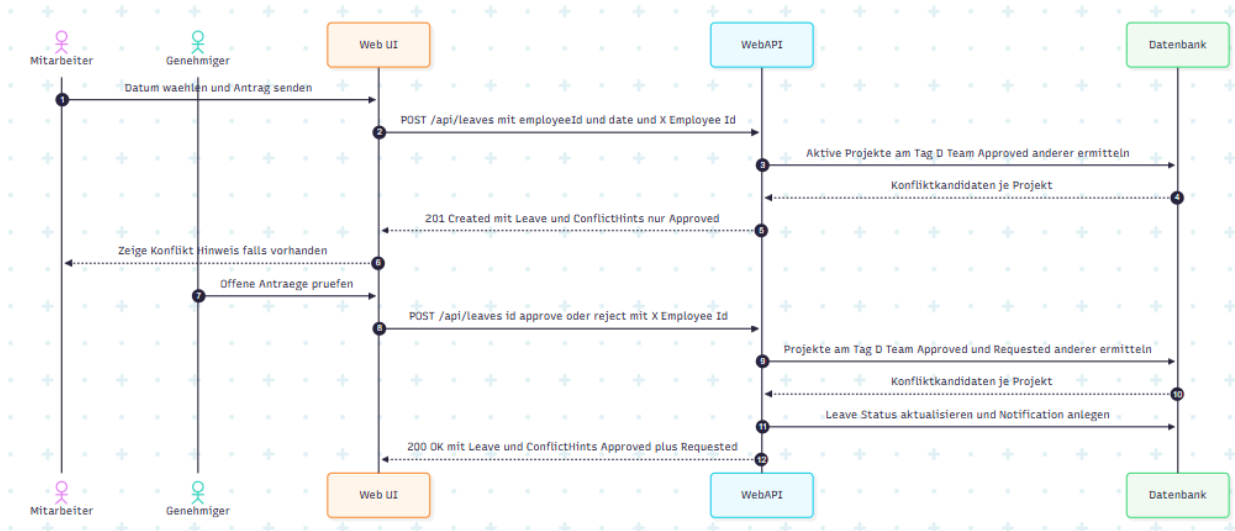
Governance & Schutzmechanismen Beispiele

- Letzter Admin kann nicht gelöscht oder zu einer Nicht-Admin-Rolle demotet werden → 409 Conflict.
- Rollenumstellung nur mit Admin-Identität → sonst 403.
- Projekt-Plausibilität: EndDate muss leer oder \geq StartDate sein → sonst 400 Bad Request.
- Eindeutigkeit: (EmployeeId, ProjectId) nur einmal (Assignments) → sonst 409.
- Admin darf in Backend fremde Inboxes lesen ([GET /api/notifications?userId=...](#)) und fremde IDs als gelesen markieren (Support-Szenarien).

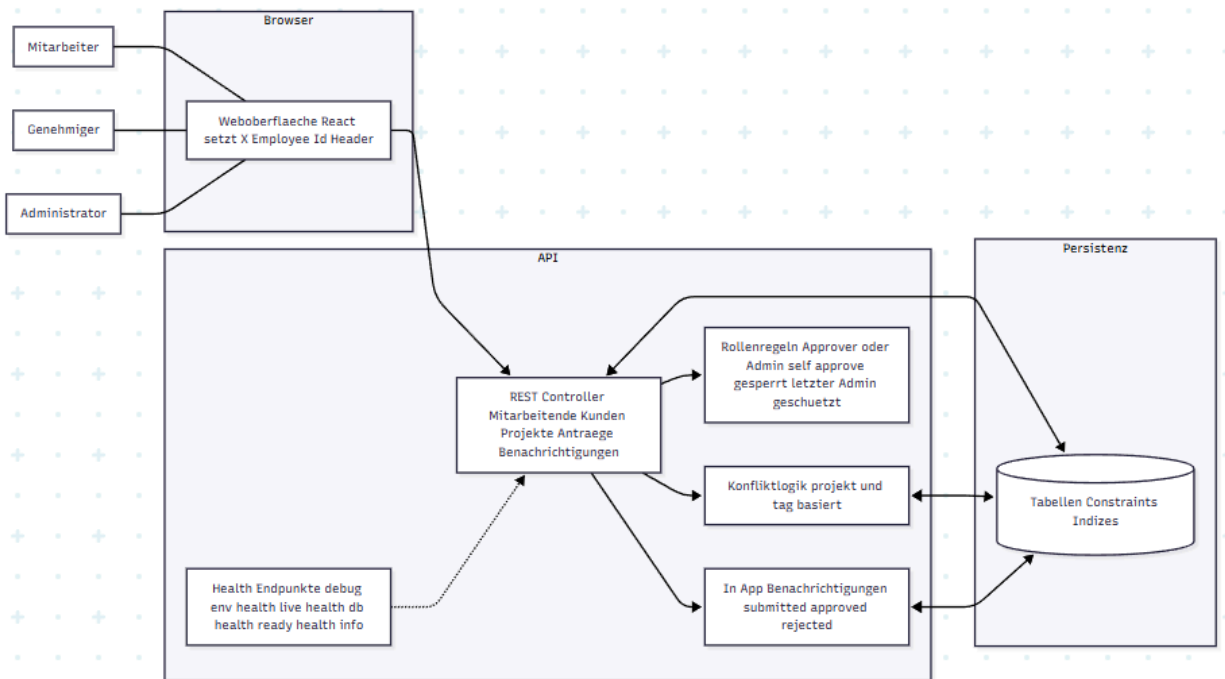
1.4 Kernereignisse

1. **Antrag erfassen** (mit Datum D): Das System ermittelt die aktiven Projekte der Person an D und liefert Konflikthinweise gegen bereits genehmigte Abwesenheiten anderer Teammitglieder.
2. **Entscheiden** (Approve/Reject): Vor der Genehmigung wird gegen genehmigte + beantragte Abwesenheiten anderer geprüft (um Überlast vorab zu erkennen).

3. **Benachrichtigen:** Der/die Antragsteller*in erhält eine In-App-Notification („eingereicht“, „genehmigt“ oder „abgelehnt“).



1.5 Systemgrenzen & Kontext



Im Umfang v1 (InScope)

- **Eintägige Urlaubsanträge:** Anlegen, Anzeigen, Approve/Reject; Status Requested/Approved/Rejected
- **Konflikthinweise (projektbezogen)**
 - Bei Antragstellung: Abgleich gegen Approved anderer Teammitglieder am selben Tag.
 - Bei Genehmigung: Abgleich gegen Approved + Requested anderer.
- **Rollen & Guards:** Employee/Approver/Admin, Self-Approve gesperrt, letzter Admin geschützt.
- **Stammdatenpflege:** Mitarbeitende, Kunden, Projekte (Start/Ende), Projektzuordnungen (Unique).
- **In-App-Benachrichtigungen:** Inbox mit submitted/approved/rejected, Mark-as-read.
- **Dev-Komfort:** Swagger, Health/Debug-Endpoints, SQLite mit Migration & Seeding.

Außerhalb des Umfangs v1 (Out of Scope)

- **Mehrtägige oder Teilzeit-Anträge** (Spannen, Halbtage/Stunden).
- **Kalender-/Kapazitätslogik:** Feiertage, Schichten, Zeitzonen, automatische Kapazitätsrechnung.
- **Externe Kanäle & Kalender:** E-Mail/Teams-Benachrichtigungen, iCal-Feeds.
- **Produktive Auth/SSO & Integrationen:**
 - **Auth/SSO:** Standard für Anmeldung über Microsoft Entra ID. Nutzer melden sich mit Firmenkonto an (Single Sign-On). Die App bekommt ein ID-Token mit Identität + Claims (E-Mail, Name, Rollen/Groups).
 - **Multi-Tenant:** Eine Installation bedient mehrere Firmen/Mandanten. Daten werden **streng getrennt** (TenantId). Jede Firma nutzt **ihr eigenes** Azure AD & Software System.
 - Mehrsprachigkeit.
- **Exporte/Reports** (CSV/iCal) und fortgeschrittene Reporting-Funktionen.

<https://asana.com/de/resources/scope-management-plan>

1.6 Explizite Annahmen & Randbedingungen

Annahmen (assumptions)

- A1 – Ein Tag je Antrag. Ein Urlaubsantrag betrifft genau einen Kalendertag.
- A2 – Projektbezug. Konflikte zählen nur in Projekten, in denen die Person am Tag *D* aktiv ist ($\text{Start} \leq D \leq \text{End}$ bzw. End leer).
- A3 – Per Team. Vergleich mit Teammitgliedern desselben Projekts, nicht mit dem ganzen Unternehmen.
- A4 – Rollen reichen. Employee / Approver / Admin decken V1 ab; Self-Approve ist verboten.

- A5 – Demo-Identität. Lokal/Dev setzen wir die Identität per **X-Employee-Id**; produktiv kommt OIDC/Azure AD.
- A6 – Ein Mandant, eine Zeitzone. Keine Feiertage/Schichten in V1.
- A7 – UI ist Demo. React-UI zeigt Kernabläufe (Request, Approve/Reject, Konflikte, Inbox), keine Enterprise-UX.
- A8 – In-App-Benachrichtigungen genügen. E-Mail/Teams kann später kommen.
- A9 – Datenmengen moderat. Kleine bis mittlere Teams; einfache Listen/Filter genügen in V1.

Beschränkungen (constraints)

- C1 – Eindeutiger Antrag. (EmployeeId, Date) ist unique → 1 Antrag je Person & Tag.
- C2 – Eindeutige Zuordnung. (EmployeeId, ProjectId) ist unique → keine Doppel-Assignments.
- C3 – Projekt-Zeitraum. EndDate ist NULL oder \geq StartDate (DB-Check).
- C4 – Server-Guards. Approve/Reject nur Approver/Admin, Self-Approve gesperrt, letzter Admin geschützt.
- C5 – Fehlercodes. Klare HTTP-Codes & kurze Texte: 400/401/403/404/409.
- C6 – Migrationen/Seeding. DB migriert beim Start; Seeding nur in Development.
- C7 – Hosting. Dev: Swagger, CORS. Prod: HTTPS/HSTS, Health-Endpoints.
- C8 – SQLite-Grenze. ORDER BY DateTimeOffset wird nicht unterstützt → Notifications DB-seitig filtern, serverseitig sortieren.
- C9 – Sicherheit Demo vs. Prod. Header-Identität ist nur für Demo zulässig; produktiv OIDC/JWT + Policies.

<https://www.linkedin.com/advice/0/how-do-you-define-project-assumptions-constraints?lang=de&originalSubdomain=de>

1.7 Beispiel Indikatoren (Erfolg messen)

Business-Wirkung

- **Durchlaufzeit Antrag → Entscheidung**
Definition: Median Zeit von Create bis Approve/Reject.
Ziel Beispiel: ↓ auf < 2 Arbeitstage.
Erhebung: Timestamps LeaveRequests.Created (implizit via Insert) → DecisionAt.
- **Konfliktarme Genehmigungen**
Definition: Anteil genehmigter Anträge **ohne** Konflikthinweis bei Genehmigung.
Ziel Beispiel: $\geq 80\%$.
Erhebung: ConflictHints-Länge bei /approve Response.
- **Vermeidung kritischer Doppel-Abwesenheiten**
Definition: Anzahl Fälle „ ≥ 2 Approved im selben Projekt am selben Tag“.
Ziel Beispiel: ↓ trendend gegen 0.
Erhebung: Query über LeaveRequests(Status=Approved) gruppiert nach (ProjectId, Date).

Nutzung & UX

- **UI-Adoption**
Definition: Aktive Nutzer/Woche (distinct Employees mit Anträgen/Entscheidungen).
Ziel: Steigend über die Einführungswochen.
Erhebung: Requests pro EmployeeId.
- **Zeit bis Notification gelesen**
Definition: Median Zeit von Notification CreatedAt bis IsRead=true.
Ziel: ≤ 1 Arbeitstag.
Erhebung: Notifications.CreatedAt vs. Mark-Read-Zeitpunkt (Serverzeit).
- **Fehlerfeedback-Quote**
Definition: Anteil Requests mit 4xx/5xx, getrennt nach 401/403/409.
Ziel: 4xx nur bei echten Regelverletzungen; 5xx ≈ 0 .
Erhebung: API-Statuscodes in Telemetrie.

<https://asana.com/de/resources/key-performance-indicator-kpi>

2) Architektur der Softwarelösung

2.1 Stil & Leitplanken

Zielbild:

Eine schlanke, wartbare REST-API (ASP.NET Core) mit klaren Verantwortlichkeiten. Persistenz via EF Core (Dev/Demo: SQLite). Eine React-UI demonstriert die Kernabläufe.

Stilprinzipien

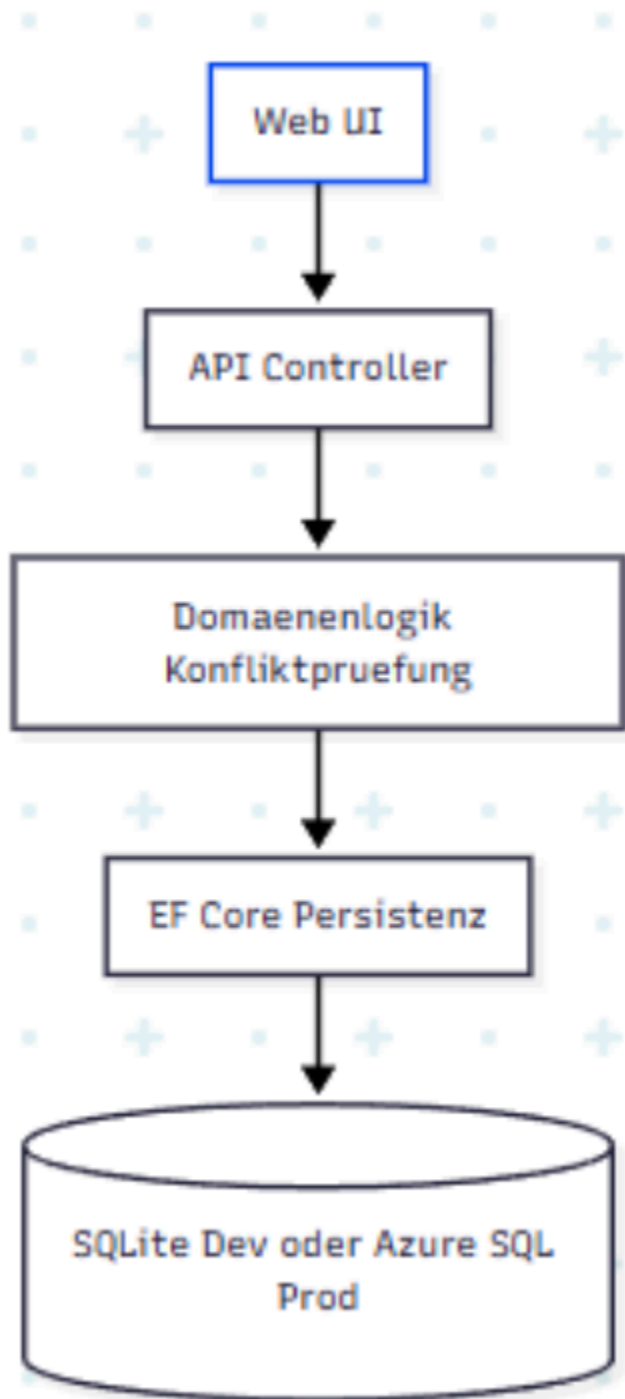
- **Einfachheit vor Vollständigkeit:** Wenige Schichten, dafür klare Regeln im Backend.
- **Serverseitige Governance:** Kritische Prüfungen (Rollen, Self-Approve-Sperre, „letzter Admin“) laufen stets im Backend.
- **Vorhersehbare DTOs & Fehlercodes:** Flache, konsistente Contracts; standardisierte 4xx/5xx-Antworten.
- **Migrationen & Health by default:** DB wird beim Start migriert; Health-Endpoints zeigen Betriebszustand.

Layers

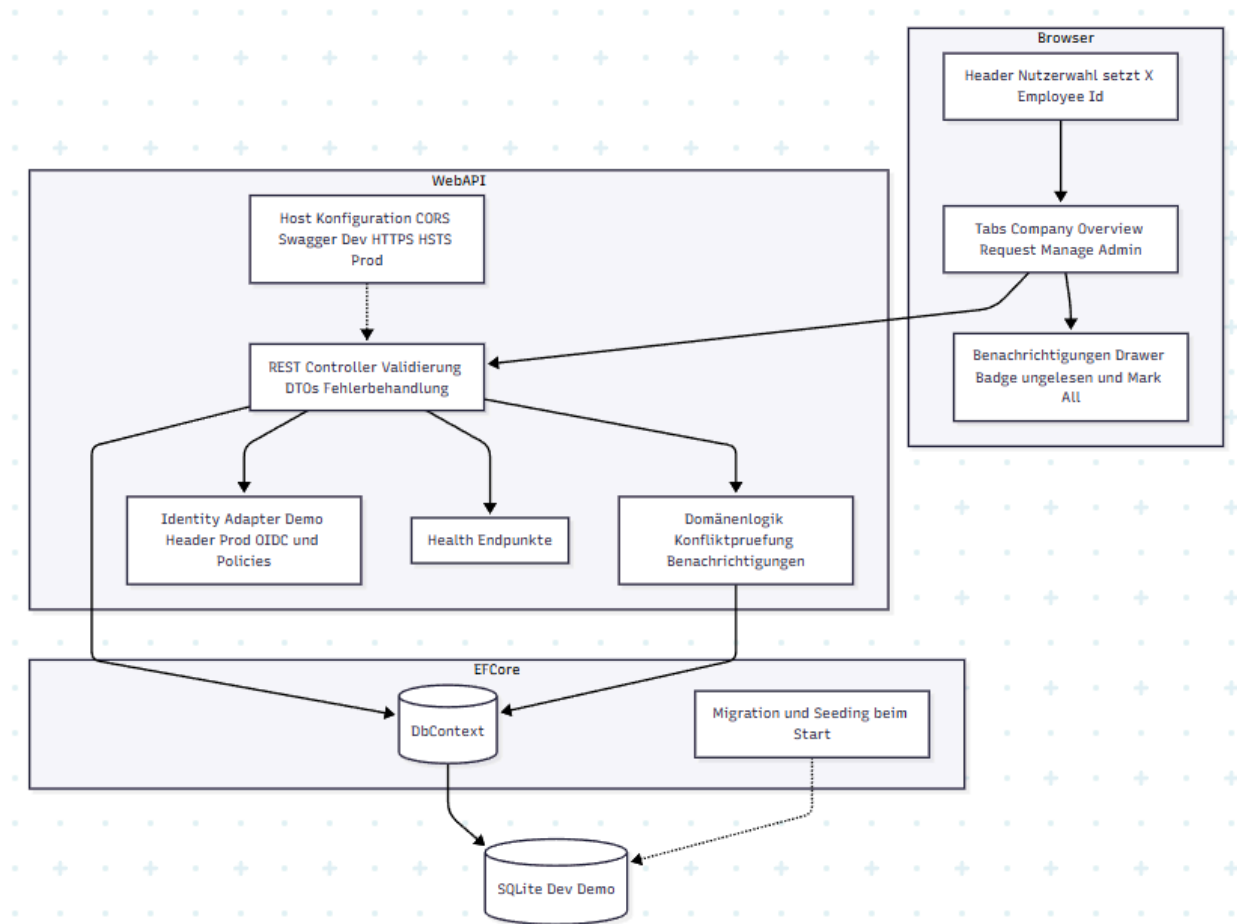
- **UI (React, optional):** steuert User-Kontext (X-Employee-Id), ruft JSON-APIs, zeigt Konflikthinweise & Inbox.
- **API-Controllers (ASP.NET Core):** Validierung, Rollenprüfungen, Orchestrierung der Domänenlogik, Fehlerhandling.
- **Domänenlogik (nah am Controller):** Konfliktprüfung, Benachrichtigungen, Guards.
- **Persistenz (EF Core):** Entitäten, Constraints/Indizes, DbContext, Migrationen.

- **Stil:** Leichtgewichtige REST-API (ASP.NET Core), Persistenz per EF Core, in Dev/Demo SQLite (Migration & Seeding beim Start).
- **UI:** Eine React-UI demonstriert die Kernabläufe (Antrag, Entscheidung, Konflikthinweise, Inbox).
- **Leitplanken:** klare Verantwortlichkeiten, vorhersagbare DTOs, serverseitige Regeln für Sicherheit/Governance, einfache Erweiterbarkeit.

Schichten-Skizze:



2.2 Hauptkomponenten & Verantwortungen



Controllers (REST)

- **EmployeesController:** CRUD Mitarbeitende (Name, JobTitle, Role lesen), Standardvalidierung „Name erforderlich“.
- **CustomersController:** CRUD Kunden, Validierung „Name erforderlich“.
- **ProjectsController:**
 - CRUD Projekte (Name, CustomerId, Start/Ende mit Check-Constraint EndDate ≥ StartDate oder NULL).
 - **Assignments:** [POST /{projectId}/assignments](#) (Zuordnen), [GET](#) (listen), [DELETE](#) (entfernen). Eindeutigkeits-Index (EmployeeId, ProjectId) verhindert Doppelzuordnung.
 - **GetAll** liefert Kunde eingebettet (UI-freundlich); [GetById](#) flach (bewusster Trade-off, README erklärt's).
- **LeavesController:**
 - [POST /api/leaves](#) (Beantragen, 1 Tag), [GET/GET {id}](#) (lesen).

- POST /{id}/approve / POST /{id}/reject: **Approver/Admin-Guard, Self-Approve-Sperre**, Auditfelder (DecisionBy, DecisionAt, DecisionComment).
- Konfliktlogik: Bei Beantragung nur gegen Approved anderer; bei Genehmigung gegen Approved + Requested anderer. Ergebnis als **conflictHints** gruppiert pro Projekt.
- **NotificationsController:**
 - GET /api/notifications?onlyUnread=...&userId=... mit Identitätsprüfung (Self erlaubt; fremde Inboxes nur Admin).
 - DB-seitige Filterung, Sortierung nach **CreatedAt** in-memory (DateTimeOffset/ORDER BY-Limit). -> SQLite-Workaround
 - POST /api/notifications/mark-read (nur eigene IDs; Admin darf fremde markieren – Support-Use-Case).

Domänenlogik

- **Konfliktprüfung:** Datum → aktive Projekte der Person → Team ohne Antragsteller:in → Leaves am D mit erlaubten Status → Gruppierung je Projekt → Liste betroffener Kolleg:innen.
- **Benachrichtigungen:** Erstellung auf Submit/Approve/Reject inkl. Actor-Metadaten.

Persistenz (EF Core / SQLite)

- Entitäten: **Employee, Customer, Project, ProjectAssignment, LeaveRequest, Notification**
- **Indizes/Constraints:**
 - **Unique:** (EmployeeId, Date) (ein Leave je Person & Tag).
 - **Unique:** (EmployeeId, ProjectId) (keine Doppelzuordnung).
 - **Check:** EndDate NULL ODER ≥ StartDate.
 - Indizes auf häufigen Pfaden (**Notifications: UserId, IsRead, CreatedAt**).
- **Migrationen beim Start, Seeding in Dev** (Stammdaten; optionale Leave-Seeds für Demo-Konflikte).

Host/Infra (Program.cs)

- **Dev:** CORS für **http://localhost:5173/3000**, Swagger-UI, Seeding.
- **Prod:** HTTPS/HSTS aktiviert, Swagger deaktiviert; Health-Endpoints aktiv.
- Health: **/debug/env, /health/live, /health/ready, /health/db, /health/info**.

2.3 Schnittstellen (auf Gruppen-Ebene)

- **REST/JSON-Ressourcen:**
`/api/employees`, `/api/customers`, `/api/projects` (+ `/assignments`),
`/api/leaves` (create/get/approve/reject), `/api/notifications` (list/mark-read)
- **Identität (Demo):** `X-Employee-Id: <GUID>` ist für Notifications & Entscheidungen obligatorisch.
- **Fehlerbilder:** Details zu Entitäten in Kap. 4.

2.4 Architekturentscheidungen & Evolutionspfad

Basiseinheit „Ein-Tages-Antrag“

Entscheidung:

- Ein Antrag entspricht genau einem Kalendertag.
- DB-Eindeutigkeit: `UNIQUE(EmployeeId, Date)`.
- Konfliktprüfung ist deterministisch (Tag D → aktive Projekte → Team → Leaves).

Wirkung:

- Minimale Geschäftsregeln, klare UX, schneller MVP.
- Vermeidet früh Komplexität (Feiertage, Teilzeit, Schichten, Zeiträume).
- Anwender können viele einzelne Tage hintereinander beantragen (Usability).

Eskalation

- Architekturänderung -> API akzeptiert (`startDate`, `endDate`); Server expandiert in einzelne Werktage.
- Datenmodell -> Weiterhin eine Zeile pro Tag (wiederverwendet die existierende Konfliktlogik). Vielleicht „BatchId“ zur Klammerung eines Zeitraums.
- **Schritte:**
 - `POST /api/leaves/range` einführen; Validierung: `start <= end`, nur Werktage.
 - Serverseitige Expansion → pro Tag `LeaveRequest` schreiben (Transaktion).
 - Konfliktprüfung pro erzeugtem Tag (identische Logik).
- Betrieb/Monitoring durch App Insights Event: „RangeCreated“ inkl. Anzahl expandierter Tage, oder Warnung bei sehr großen Zeiträumen (>20 Werktage).

Konflikte sind projektbezogen

Entscheidung:

- Ein Konflikt entsteht nur, wenn am selben Tag im selben aktiven Projekt bereits Leaves anderer Teammitglieder vorliegen.
- Bei Genehmigung wird gegen Approved + Requested geprüft; bei Beantragung nur gegen Approved.

Wirkung:

- Fachlich relevant ist die Team-Kapazität im gleichen Projekt.
- Klarer Hinweis: pro Projekt gruppiert (ConflictHints).

Risiken

- Projektübergreifende Abhängigkeiten bleiben unsichtbar (Portfolio-Sicht).

Eskalation

- **Option A – Portfolio-Indikator (separat):**
 - Approver sehen beim Genehmigen zusätzlich zur projektbezogenen Konfliktprüfung eine übergreifende Kapazitätsampel für den Tag (z. B. „Portfolio-Kapazität angespannt“). Das hilft, Entscheidungen *am Ort des Geschehens* zu verbessern.
 - Zusätzlicher Endpunkt `/api/capacity/portfolio?date=...` liefert eine aggregierte Sicht je Kunde/Produktlinie.
 - SQL-View oder Materialized View (Azure SQL) zur Voraggregation (z. B. „Abwesenheiten je Projektfamilie“).
 - UI: Separater Hinweis im Approve-Dialog („Portfolio-Kapazität angespannt“).
- **Option B – Reporting/BI:**
 - Trends, Saisonalität, kritische Teams und historische Muster erkennen. Das ist nicht für die momentane Entscheidung im Approve-Dialog gedacht, sondern für Planung, Controlling und Management (wöchentliche/monatliche Auswertungen).
 - Export: Genehmigte Leaves regelmäßig in Azure Data Lake/Blob oder Power BI Dataset schreiben.

Serverseitige Guards und Login

Entscheidung:

- Rollenprüfung (Approver/Admin), Verbot Self-Approve/-Reject, Schutz „letzter Admin“ sind serverseitig erzwungen (403/409). Das sind Sicherheits- und Geschäftsregeln, die immer auf dem Server geprüft werden, nicht nur in der Oberfläche.

Wirkung:

- Auch wenn jemand die UI ändert oder direkt HTTP-Aufrufe schickt – der Server lässt unzulässige Aktionen nicht durch.
- Eindeutige, konsistente Fehlerbilder; Revision sicher.
- **Approver** können nur fremde Anträge entscheiden – keine „Abstimmung mit sich selbst“.
- **Admin-Governance** bleibt erhalten – niemand „verliert“ aus Versehen den letzten Admin.

Eskalation

- Login: Nutzer melden sich über Microsoft Entra ID an (Single Sign-On). Die Web-App bekommt ein JWT-Token mit Nutzer-ID und Rollen/Ansprüchen.
- API-Schutz: Die API prüft jedes eingehende Token automatisch
- „Approver/Admin“-Zugriff als Policy.
- Governance: Letzter-Admin-Schutz bleibt eine serverseitige Regel (unabhängig vom Login). Änderungen an Rollen sind nur Admins erlaubt.
- Betrieb/Transparenz: Application Insights protokolliert fehlgeschlagene Logins, verbotene Aufrufe und ungewöhnliche Muster.

Service-Layer (ConflictService, NotificationService)

Entscheidung:

Jetzt ist Logik controller-nah. Perspektivisch Services für klare Verantwortungen & Tests.

Wirkung:

- wenig Boilerplate, schnelle Änderung.

Risiken:

- Wachsende Codebasis → Wiederholungen im Controller.

Eskalation:

- Schnittstellen: **IConflictService** (Projekte am Tag D → Team → Konflikte), **INotificationService** (erzeugen/listen/mark-read, später Outbox).
- DI & Tests: Unit-Tests auf Service-Ebenen, Controller nur Orchestrierung/DTO-Mapping.
- Wartbarkeit: Open-Closed (neue Konfliktregeln isoliert erweiterbar).

Benachrichtigungen extern

Entscheidung:

Jetzt nur In-App (Inbox). Kein externer Versand (Mail/Teams) im Requestfluss.

Risiken:

- Stakeholder außerhalb der App bleiben uninformiert; synchroner Versand würde Requests verlängern.

Eskalation:

- Outbox-Tabelle, die „Versandaufträge“ sammelt + Warteschlange in der Cloud. Der Hintergrunddienst nimmt neue Aufträge von dort ab (Azure Service Bus) + Worker -> kleines Programm im Hintergrund, das Nachrichten aus der Queue holt, z. B. über Microsoft Graph E-Mails/Teams sendet, und den Auftrag als erledigt markiert.
- API schreibt Outbox in derselben Transaktion; Worker liest, sendet via Microsoft Graph (Mail/Teams), markiert verarbeitet.

Pagination & Caching (Listen stabilisieren)**Entscheidung:**

Derzeit vollständige Listen ohne serverseitige Pagination; Stammdaten ungecached.

Wirkung:

- Simpler Start; bei wachsenden Daten potenziell träge.

Risiken:

- Lange Ladezeiten/hoher Payload (Notifications/Leaves).

Eskalation:

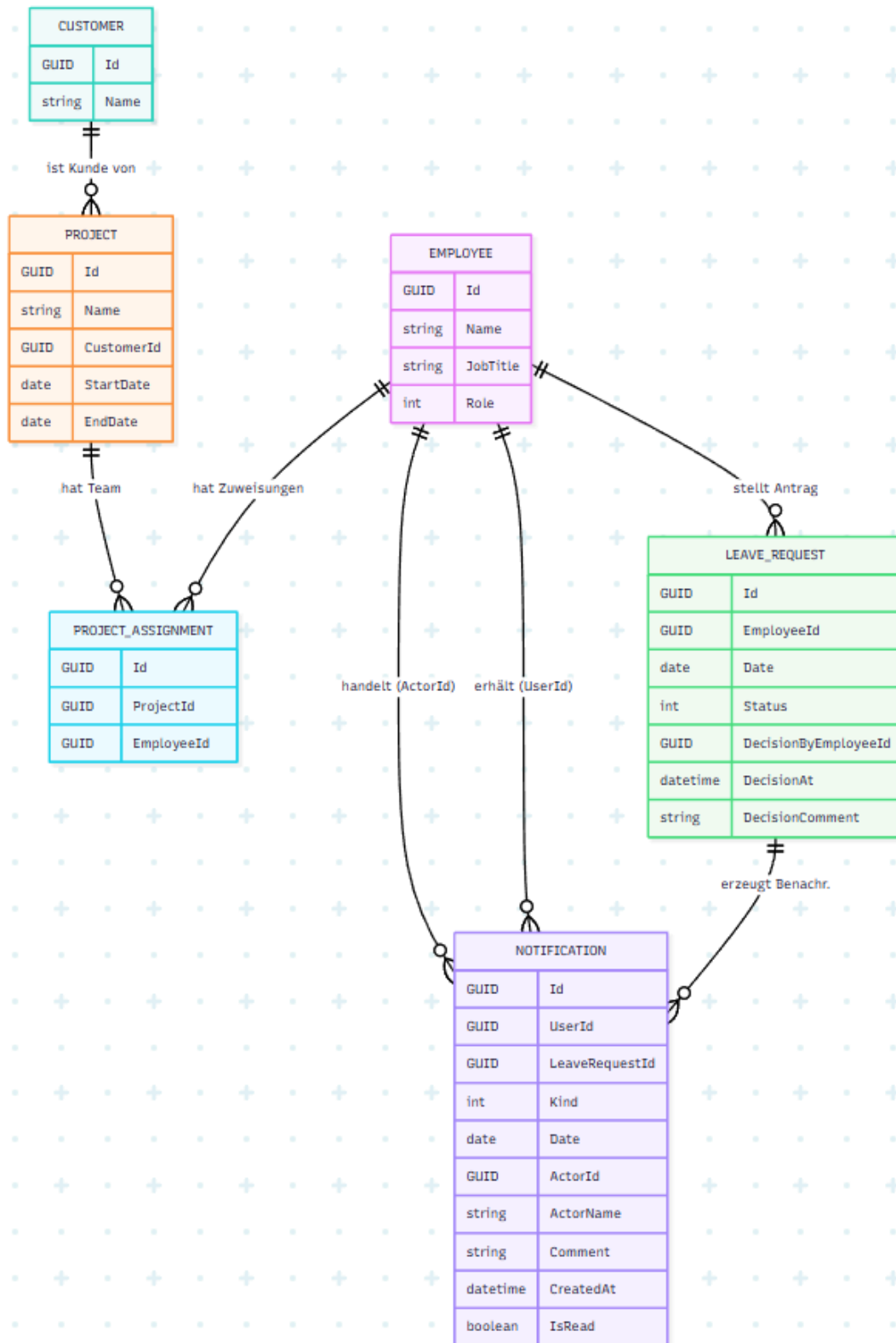
- API: `?skip=&take=` auf `/api/notifications` und `/api/leaves`, sinnvolle Defaults (50/100).
- Client: „Weitere laden“/Infinite-Scroll.
- Caching: Stammdaten mit E-Tag/If-None-Match; optional Redis für Hot-Sets.
- Indizes: sicherstellen, dass Filter/Sort-Spalten indexiert sind.
- Telemetry: p95-Latenzen der Listen, Bytes-pro-Response, Fehlerquote.

4) Persistenz & Datenmodell

4.1 Informationsmodell (Kernentitäten)

- **Employee** – Name, JobTitle, Role
- **Customer** – Name
- **Project** – Name, CustomerId, StartDate, EndDate?
- **ProjectAssignment** – (EmployeeId, ProjectId)
- **LeaveRequest** – EmployeeId, Date, Status, DecisionBy?, DecisionAt?, DecisionComment?
- **Notification** – UserId, LeaveRequestId, Kind, Date, ActorId?, ActorName?, Comment?, CreatedAt, IsRead

Bezugsdiagramm (ERD)



4.2 Constraints

- Pro Mitarbeiter und Datum darf es nur einen Leave geben.
- Pro Mitarbeiter und Projekt darf es nur eine Assignment-Zuordnung geben.
- Projekte haben die Bedingung: $EndDate \geq StartDate$.

4.3 Probleme / Grenzen:

1. **Skalierung & Parallelität:**
 - SQLite eignet sich nur für Entwicklung
 - Keine echte **Connection-Pooling**-Strategie.
 - Schreibkonflikte bei mehreren gleichzeitigen Requests.
2. **Datenqualität:**
 - Keine Soft Deletes: Einträge verschwinden endgültig, Nachvollziehbarkeit fehlt.
 - Löscht man ein Projekt, bleiben ggf. Leaves bestehen → Inkonsistenzen möglich.
 - Zeit- und Datumsfelder: als string gespeichert → UI und Persistenz können auseinanderlaufen (Zeitzonenproblem).
 - Fehlende **Audit-Logs** → wer hat wann einen Antrag genehmigt/abgelehnt?
 - Kein **historischer Verlauf** → Statusänderungen überschreiben sich.
3. **Multi-Tenancy fehlt:**
 - In v1 ist nicht vorgesehen, wenn das Projekt in Prod freigegeben, mehrere Kundenmandanten isoliert zu betreiben -> Daten von Kunde A können mit Kunde B kollidieren.

4.4 Zukunft Vision

Datenbankwahl & Betrieb

- Wechsel von SQLite in dev → SQL Server oder PostgreSQL in prod.
 - Hochverfügbarkeit, automatische Backups.
 - **Elastic Pools** für mehrere Mandanten (Kostenoptimierung).
 - **Transparent Data Encryption (TDE)** für Datensicherheit.
- Für sensible Tabellen (z. B. *Employees* bei **Rollenänderungen**, *Projects* bei Laufzeit) plane ich eine **Concurrency-Spalte** so wenn zwei Personen gleichzeitig ändern, schlägt die zweite Speicherung mit „Concurrency conflict“ fehl → die UI zeigt „Bitte Seite aktualisieren“.

Migration & Schema-Management

- Einsatz von **Entity Framework Core Migrations** (CI/CD integriert) -> Dev/Prod-Trennung über Azure DevOps Pipelines

Historisierung & Nachvollziehbarkeit

- **Audit-Tabelle**
- **Event Sourcing:** Speicherung jeder Statusänderung als Event → für Reporting & Replay.

Erweiterungen & Domänenlogik

- **Notifications:** eigene Tabelle UserNotifications, triggerbasiert oder via ServiceBus.E-Mail/Teams Notifications werden asynchron über Outbox + Queue versendet. So bleibt der fachliche Zustand sofort konsistent; externe Zustellung darf nachlaufen
- **Logs:** zentrale Ablage in Azure Application Insights.
- **Multi-Tenant-Isolation:**
 - Spalte TenantId in allen Entities.
 - Abfragefilterung über EF Core Global Query Filter.
 - Pro Tenant eigene Datenbank (z.B. mit Azure SQL Elastic Pool).

Skalierung

- **Horizontale Skalierung**
 - **API** auf Azure App Service/Kubernetes, mehrere Instanzen, stateless.
 - **Worker** (Azure Functions/.NET Worker) skaliert nach Queue-Last.
 - **Service Bus** (Queue/Topic) puffert Lastspitzen stabil weg.
- **Datenbank-Skalierung**
 - **Azure SQL:** Größere SKU, **Read-Replica** für BI, **Partitionierung** großer Tabellen (z. B. *Leaves*, *Notifications* nach Jahr/Tenant).
 - Ältere Daten **archivieren** (Kaltpfad), nur aktuelle Jahrgänge „heiß“ halten.

Sicherheit

- **Transparent Data Encryption (TDE)** für alle Datenbanken.
- Retention-Policies (z. B. „Notifications 180 Tage“ → **Archiv**-Tabelle/Blob).
- Pseudonymisierung/Anonymisierung der Audit-Daten nach Vorgabe (DSGVO).

Beobachtbarkeit & SLOs

- **Application Insights:** p95-Latenzen, Throughput, Fehlerraten, Slow Queries.
- **Alerts:** DLQ > 0, DB-CPU > x%, 5xx-Rate > y, Latenz > SLO.

Chaos/Load-Tests auf Kernpfaden (Create/Approve/Reject, Notifications-Listen).

7) Skalierung & Performance

4.1 Datenbank

- **Indexe** auf wichtige Spalten (z. B. Mitarbeiter-ID, Projekt-ID, Datum). → Suchen und Vergleichen gehen schneller.

4.2 API

- **Batch-Endpunkte** statt viele kleine Abfragen. Beispiel: Statt 100 Mal „welche Mitarbeitenden sind in Projekt X?“ → eine Abfrage liefert alle Projekte **inkl. Zuweisungen**.
- Optional: langfristig **GraphQL**, wo das Frontend genau die Daten anfordern kann, die es braucht.

4.3 Caching

- Häufig gebrauchte Daten (z. B. Stammdaten wie Kunden oder Projekte) werden im **Cache** (z. B. Redis) für einige Minuten gespeichert → reduziert Last auf die Datenbank.

4.4 Echtzeit-Kommunikation

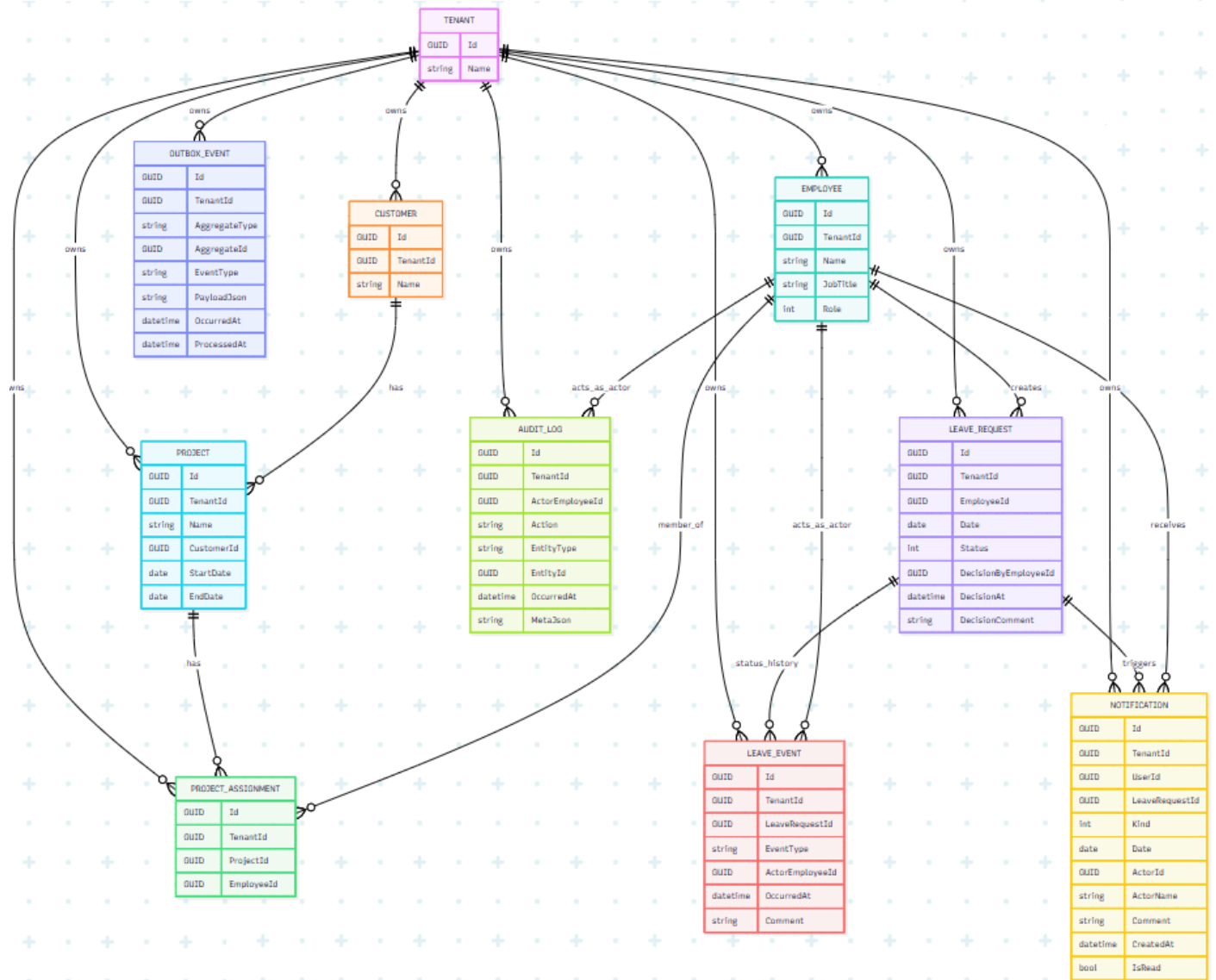
- Statt Polling, SignalR oder WebSockets nutzen -> Benachrichtigungen landen sofort im Frontend, ohne dass es ständig nachfragen muss.

4.5 Architektur & Betrieb

- **Containerisierung** (Docker, Kubernetes) → die Anwendung kann auf mehreren Servern parallel laufen („horizontale Skalierung“).
- **Monitoring & Logging** (z. B. Prometheus, Grafana, strukturierte Logs) → wird sofort gesehen, wo Engpässe oder Fehler auftreten.

4.6 Beziehungsdigramm (ERD)

- **Mandantenfähigkeit (TENANT)**: Mehrere Firmen/Teams in einer Installation sauber trennen.
- **Status-Historie (LEAVE_EVENT)**: Jede Statusänderung eines Antrags wird als Ereignis protokolliert.
- **Outbox-Muster (OUTBOX_EVENT)**: Fachereignisse verlässlich für Integrationen & Benachrichtigungen festhalten.
- **Audit-Trail (AUDIT_LOG)**: speichert technische/administrative Aktionen (z. B. „Employee.Role updated“, „Project deleted“), Actor, Entity, Zeit, Meta (JSON).
- **Konsequente Indizes & Keys**: Eindeutigkeit und schnelle Abfragen auch bei großen Datenmengen.



4.7 Komponenten-Diagramm

1. Browser → Gateway → API

Alle HTTP-Requests gehen über **Ingress/API-Gateway** an eine **horizontale Menge API-Pods** (Load-Balancing).

→ Hohe Verfügbarkeit & einfache **horizontale Skalierung** (mehr Pods bei Last).

2. API nutzt Cache und DB richtig

- **Redis Cache** für häufige Reads (z. B. Kunden/Projekte).
- **Managed SQL** als „Source of Truth“ für Schreib-/Lesevorgänge mit korrekter Transaktionssicherheit.

→ **Weniger DB-Last, schnellere Antwortzeiten** (4.3 Caching, 4.1 Indexe).

3. Outbox-Events sichern Fachereignisse

Bei wichtigen Aktionen (z. B. „Urlaub genehmigt“) schreibt die API zusätzlich einen Datensatz in **Outbox** (in derselben DB-Transaktion wie der Business-Write).

→ **Verlustfrei**, auch wenn Downstream gerade nicht erreichbar ist (4.6 Outbox-Muster).

4. Worker entkoppelt Integrationen

Der **Background Worker** liest die **Outbox** in eigenem Takt, triggert **Notifier** (E-Mail, Webhooks) und **Realtime-Push** via **SignalR Hub**.

→ Lastspitzen werden geglättet; API bleibt schlank und schnell (kein Warten auf externe Systeme).

5. SignalR Hub → Echtzeit statt Polling

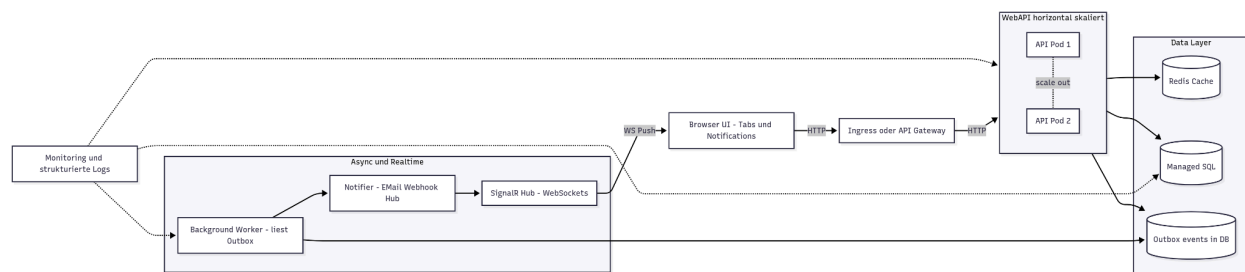
Der **Hub** pusht Benachrichtigungen direkt in den Browser (WebSockets).

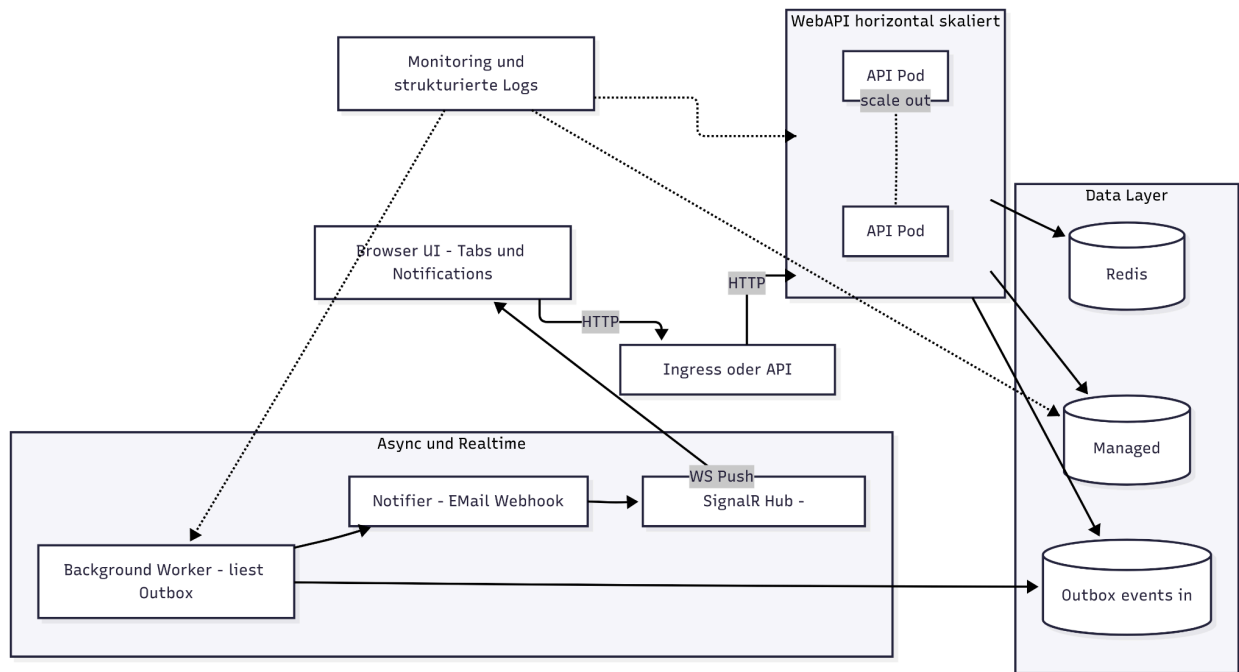
→ Sofortige Updates; **kein Polling**, dadurch weniger Netz-/API-Last (4.4 Echtzeit).

6. Observability überall

Monitoring & strukturierte Logs betrachten **API, DB und Worker**.

→ Du siehst Bottlenecks (Queries, Throughput, Latenzen) sofort; schnellere Fehleranalyse (4.5 Monitoring & Logging).





**Die beiden Diagramme stellen dieselbe Architektur dar – es gibt keine inhaltlichen Unterschiede, weil Bei der zweiten Variante war es schwieriger, alle Pfeile sauber auszurichten