Interactive Classroom

Interactive Classroom ist eine Mehr-Service-Anwendung für digitale Kurse mit:

- Login, Registrierung und Gastzugang
- JWT-basierter Authentifizierung mit Refresh-Token
- Kursen (`Environments`) und darin enthaltenen Sessions
- kollaborativem Whiteboard mit Hocuspocus und Y.js
- Session- und Board-Realtime
- persistenter Speicherung in PostgreSQL

Schnellstart

Ein frischer Clone kann lokal direkt gestartet werden:

- macOS mit Colima: `./scripts/start.sh`
- Linux / Windows / Docker Desktop: `docker compose up --build`

Danach ist die Anwendung unter `http://localhost` erreichbar.

Typische Entwicklungs-URLs:

- App: `http://localhost`
- Traefik-Dashboard: `http://localhost:8080`
- Collab-Health: `http://localhost:1234/health`
- pgAdmin: `http://localhost:5050`

Optional:

- `.env.example` nach `.env` kopieren, falls du Defaults überschreiben willst
- ohne `.env` funktionieren die lokalen Dev-Defaults bereits

Diagramme

Zusätzliche Architektur- und Ablaufbilder liegen im Ordner `images/` und sind auch direkt im README eingebunden:

- Projektarchitektur: images/PROJEKT_ARCHITEKTUR.svg
- JWT-/Access-Flow: images/jwt-auth-flow.svg
- Datenbankübersicht: images/liveclass_board_database.svg

Was aktuell vorhanden ist

Der aktuelle Stand ist kein reines Backend-Projekt mehr, sondern ein komplettes System mit Frontend, API, Realtime und Whiteboard.

Frontend

Das React-Frontend in `apps/web` enthält aktuell:

- Login-Seite mit drei Modi:

    - Anmelden
    - Registrieren
    - Gastzugang
- geschützte Routen für Dashboard und Board
- automatischen Silent-Login beim App-Start über Refresh-Token
- Dashboard für:
    - eigene Kurse anzeigen
    - Kurs erstellen
    - Kurs per Code beitreten
    - Sessions pro Kurs anzeigen
    - Sessions öffnen, löschen oder verlassen
- Board-Seite für:
    - Session- und Kurs-Kontext anzeigen
    - Shared Boards und persönliche Boards nutzen
    - Mitgliederliste und aktives Mitglied anzeigen
    - Session öffnen und schließen
    - aktives Mitglied manuell oder zufällig setzen
    - Board als persönliche Kopie übernehmen
    - PDF-Export
    - QR-Code zum Beitritt
    - Zoom, Undo, Redo
    - Bilder einfügen
    - Mehrseiten-Notizbuch pro Board
    - Seiten anlegen und löschen
    - Hintergrundtyp und Ausrichtung pro Board / Seite

Backend / API

Die Express-API in `apps/api` enthält aktuell:

- Authentifizierung
- JWT-Ausgabe für API und Collab-Zugriff
- Refresh-Token-Verwaltung in PostgreSQL
- Kurs-/Environment-Verwaltung
- Session-Verwaltung
- Realtime-Stream für Session-Updates
- Berechnung und Synchronisation von Board-Schreibrechten
- Verwaltung von Shared Boards und persönlichen Board-Kopien

Whiteboard / Realtime

Der Collab-Service in `apps/collab` enthält aktuell:

- Hocuspocus WebSocket-Server
- Y.js-Synchronisation
- serverseitige JWT-Prüfung für Board-Dokumente
- Snapshot-Persistenz in PostgreSQL

- relationale Projektion für Board-Daten
- Redis-Backplane für verteilte Kollaboration

Architektur

Services

- `apps/web`: React/Vite-Frontend
- `apps/api`: Express-API für Auth, Kurse, Sessions und Session-Realtime
- `apps/collab`: Hocuspocus/Y.js-Service für das Whiteboard
- `db`: PostgreSQL
- `redis`: Realtime-Backplane für Hocuspocus
- `traefik`: Reverse Proxy
- `migrate`: Prisma-Migrationen beim Start
- `pgadmin`: optionale DB-Oberfläche für Entwicklung

Laufzeitbild

Zusätzlich zum Mermaid-Überblick gibt es eine visuelle Gesamtübersicht als SVG:

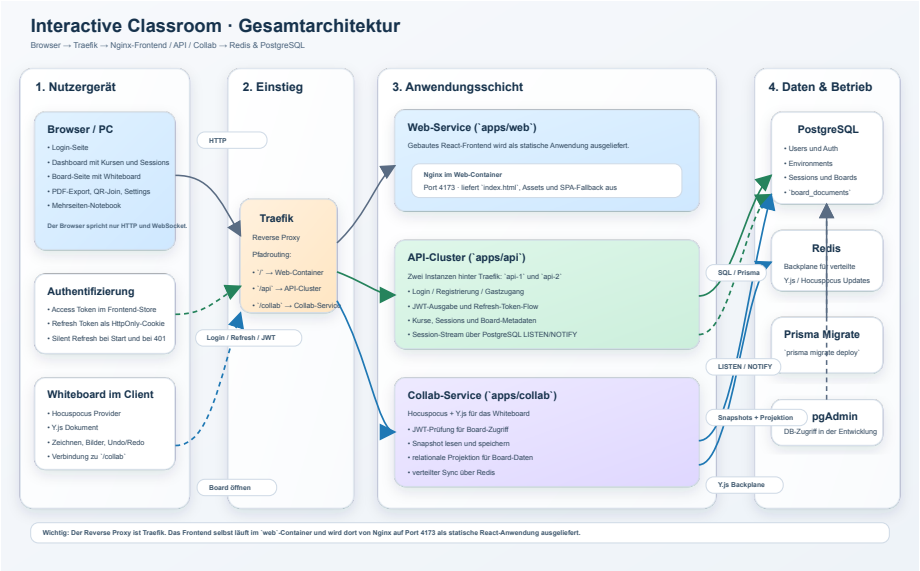

Figure 1: Projektarchitektur

```
flowchart LR
    Browser -->|HTTP| Traefik
    Browser -->|REST + Stream| API
    Browser -->|WS Y.js| Collab
```

```
Traefik --> API1[api-1]
Traefik --> API2[api-2]
Traefik --> WEB[web]

API1 --> Postgres[(PostgreSQL)]
API2 --> Postgres
API1 -->|LISTEN/NOTIFY| Postgres
API2 -->|LISTEN/NOTIFY| Postgres

Collab --> Postgres
Collab --> Redis[(Redis)]
```

Authentifizierung

Die Anwendung verwendet ein Access-/Refresh-Token-Modell:

- `POST /api/auth/login`, `register` oder `guest` liefern ein `accessToken`
- der Access Token wird im Frontend im Zustand gehalten
- der Refresh Token wird serverseitig als `HttpOnly`-Cookie gesetzt
- beim Start der App wird `POST /api/auth/refresh` aufgerufen
- bei `401` versucht das Frontend automatisch genau einen Refresh und wiederholt den Request

Das bedeutet:

- API-Aufrufe laufen mit `Authorization: Bearer <token>`
- der Benutzer bleibt nach einem Reload eingeloggt, solange der Refresh-Token gültig ist
- Logout widerruft den Refresh-Token und leert die Session im Frontend

Der Ablauf ist zusätzlich als Diagramm dokumentiert:

Fachmodell

Kurse / Environments

Ein `Environment` ist der übergeordnete Kursraum.

Ein Environment hat:

- einen eindeutigen `code`
- einen Titel
- optionale Bild-URL
- Rollen- und Join-Einstellungen
- Mitglieder
- Board-Templates
- mehrere Sessions

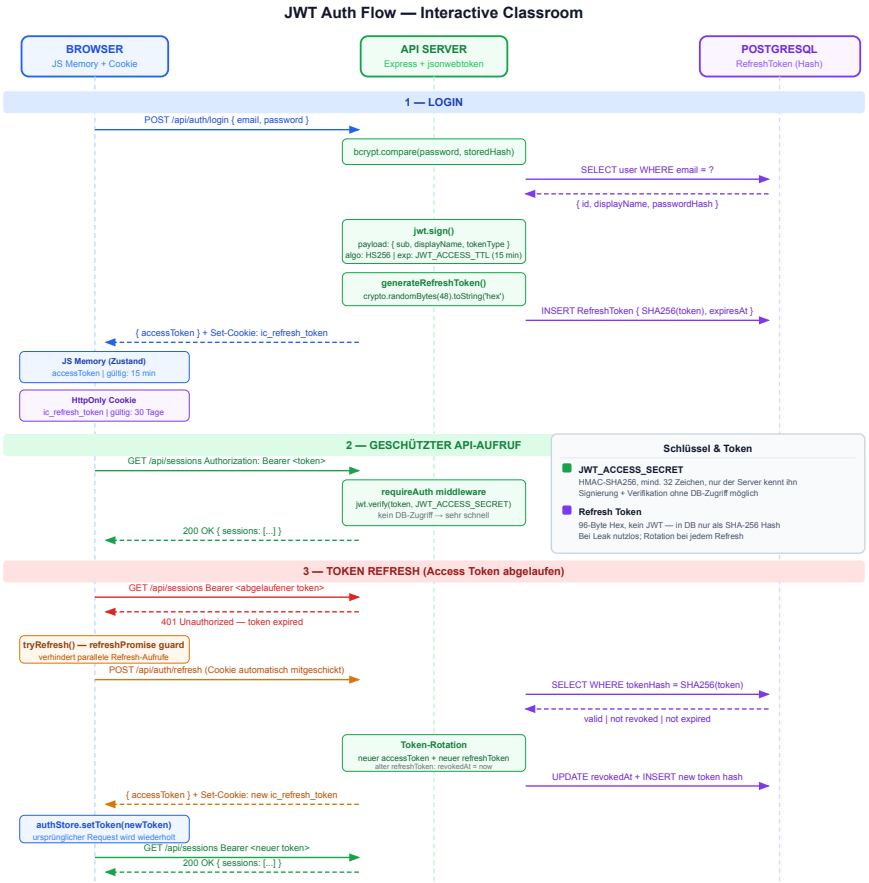

Figure 2: JWT Auth Flow

Sessions

Eine Session ist eine konkrete Durchführung innerhalb eines Kurses.

Eine Session hat:

- einen eindeutigen `code`
- einen Status: `OPEN`, `CLOSED`, `ENDED`
- Mitglieder
- ein Host-Mitglied
- optional ein aktives Mitglied
- Spaces und Boards

Boards

Die Board-Struktur ist inzwischen deutlich größer als in der alten README:

- pro Session gibt es Shared-Bereiche
- Teilnehmer können persönliche Board-Kopien besitzen
- Boards haben Metadaten wie:
  - `documentName`
  - `orientation`
  - `backgroundTemplate`
  - `width`
  - `height`
- innerhalb eines Boards existiert im Frontend ein Mehrseiten-Notizbuch
- Y.js-Zustand wird in `board_documents` gespeichert
- lesbare Projektionen liegen in relationalen Tabellen

Rechtemodell

Session-Schreibrechte

Im Shared-Bereich gilt aktuell:

- die Session muss `OPEN` sein
- der Host darf schreiben
- optional darf genau ein aktives Session-Mitglied schreiben

Diese Regel wird doppelt abgesichert:

- in der API für Status, Antwortdaten und UI
- im Collab-Service direkt vor eingehenden Y.js-Updates

Persönliche Boards

Persönliche Boards sind an ein Session-Mitglied gekoppelt. Dort arbeitet der Besitzer in seinem eigenen Bereich. Shared Boards können außerdem als persönliche Kopie übernommen werden.

Realtime

Session-Realtime

Session-Daten laufen nicht über Y.js, sondern über einen API-Stream:

```
sequenceDiagram
    participant Browser
    participant API
    participant Postgres

    Browser->>API: GET /api/sessions/:code/stream
    API->>Browser: initiales Session-Snapshot
    API->>Postgres: LISTEN session_updates
    Postgres-->>API: NOTIFY
    API-->>Browser: neues Session-Snapshot
```

Board-Realtime

Das Whiteboard selbst läuft über Hocuspocus und Y.js:

```
sequenceDiagram
    participant Browser
    participant Collab
    participant Redis
    participant Postgres

    Browser->>Collab: WebSocket connect + JWT
    Collab->>Postgres: lade Snapshot aus board_documents
    Collab->>Redis: verteile Y.js-Updates
    Browser->>Collab: Zeichnen / Bild / Seiten-Änderung
    Collab->>Postgres: speichere Snapshot
    Collab->>Postgres: aktualisiere Projektion
```

Hocuspocus, WebSocket und Y.js-Synchronisation

Die eigentliche Whiteboard-Synchronisation läuft nicht über klassische REST-Requests, sondern über einen bidirektionalen WebSocket-Kanal zwischen Browser und Collab-Service.

Einfach erklärt:

- das Frontend erstellt lokal ein Y.js-Dokument
- dieses Dokument wird mit einem Hocuspocus-Provider an den Collab-Service gebunden
- dabei wird ein WebSocket auf `/collab` geöffnet
- der Server lädt den zuletzt gespeicherten Board-Snapshot aus PostgreSQL
- danach werden Änderungen laufend in beide Richtungen synchronisiert

Bidirektional bedeutet hier:

- Client -> Server: lokale Änderungen wie Zeichnen, Bilder, Seiten oder Verschieben
- Server -> Client: initialer Snapshot und Updates anderer verbundener Benutzer

Das ist wichtig, weil ein kollaboratives Whiteboard nicht nur Daten senden, sondern gleichzeitig auch fremde Änderungen empfangen muss.

Wo wird der WebSocket geöffnet? Im Frontend wird der WebSocket nicht manuell mit `new WebSocket(...)` geöffnet, sondern über den Hocuspocus-Provider.

Der Ablauf ist:

- das Frontend erstellt ein `Y.Doc`
- daraus werden gemeinsame Datenstrukturen wie `pages` und `elements` gelesen
- der `HocuspocusProvider` verbindet dieses Dokument mit dem Collab-Service

Konzeptionell:

```
Browser
  -> Y.Doc
  -> HocuspocusProvider
  -> ws://localhost/collab
  -> Traefik
  -> Collab-Service
```

Traefik routet `/collab` an den Hocuspocus-Service weiter. Dadurch bleibt nach außen nur eine einheitliche URL sichtbar.

Wie läuft die Synchronisation beim Verbinden ab? Wenn ein Benutzer ein Board öffnet, passiert vereinfacht Folgendes:

1. das Frontend erzeugt ein lokales Y.js-Dokument
2. der Provider verbindet sich über WebSocket mit dem Collab-Service
3. der Access-Token wird beim Verbindungsaufbau mitgegeben
4. der Server prüft Token und Board-Berechtigung
5. der Server lädt den letzten gespeicherten Snapshot aus `board_documents`
6. der Snapshot wird in das verbundene Y.js-Dokument synchronisiert
7. danach laufen nur noch inkrementelle Änderungen über den offenen WebSocket

Man kann sich das wie einen gemeinsamen verteilten Dokumentzustand vorstellen:

```
Postgres Snapshot -> Collab -> Y.Doc im Browser
```

Wie kommen Änderungen von einem Benutzer zu allen anderen? Wenn ein Benutzer zeichnet oder ein Element verschiebt:

1. das Frontend ändert lokal das Y.js-Dokument
2. Y.js erzeugt daraus ein Update
3. dieses Update wird über den offenen WebSocket an Hocuspocus gesendet
4. Hocuspocus verteilt das Update an andere verbundene Clients
5. deren Y.js-Dokumente übernehmen die Änderung
6. die React-Oberfläche rendert den neuen Zustand neu

Das heißt:

- der Benutzer arbeitet immer zuerst auf seinem lokalen Dokument
- die Synchronisation verteilt nur die Zustandsänderungen
- alle Teilnehmer konvergieren auf denselben Board-Zustand

Warum ist das bidirektional? Ein normaler HTTP-Request ist eher ein Anfrage-Antwort-Muster:

- Client sendet Anfrage
- Server sendet Antwort
- Verbindung ist danach im Normalfall wieder fertig

Beim Whiteboard reicht das nicht aus. Der Server muss jederzeit neue Updates an den Browser pushen können, ohne dass der Browser ständig pollen muss.

Darum wird ein WebSocket verwendet:

- eine einzige dauerhafte Verbindung bleibt offen
- beide Seiten können jederzeit Nachrichten schicken
- der Kanal ist full-duplex, also wirklich bidirektional

Wie wird Authentifizierung und Zugriff abgesichert? Der Collab-Service akzeptiert nicht einfach jede WebSocket-Verbindung.

Beim Verbinden:

- wird der Access-Token geprüft
- wird aus dem Dokumentnamen das Ziel-Board bestimmt
- werden die Schreib- oder Leserechte für dieses Board aus der Datenbank geprüft

Während die Verbindung aktiv ist, werden eingehende Nachrichten zusätzlich erneut gegen die aktuellen Rechte geprüft. Dadurch kann eine Verbindung auch read-only sein, wenn die Session zwar sichtbar, aber nicht beschreibbar ist.

Wie wird der Zustand gespeichert? Der Collab-Service speichert nicht jeden einzelnen Mauspunkt als eigene SQL-Zeile im Hauptpfad der Synchronisation.

Stattdessen:

- hält Hocuspocus den aktiven Zustand im Y.js-Dokument
- speichert den binären Snapshot dieses Dokuments periodisch in `board_documents`
- aktualisiert zusätzlich eine relationale Projektion für lesbare Auswertungen

Dadurch entstehen zwei Ebenen:

- Y.js-Snapshot fuer die schnelle kollaborative Wiederherstellung
- relationale Tabellen fuer Abfragen, Projektion und Analyse

Warum ist dieser Ansatz sinnvoll? Die Kombination aus Hocuspocus, WebSocket und Y.js hat mehrere Vorteile:

- sehr geringe Latenz bei kollaborativen Änderungen
- kein ständiges Polling nötig
- mehrere Clients können gleichzeitig arbeiten
- Konflikte werden auf Dokumentebene über Y.js zusammengeführt
- persistente Wiederherstellung über gespeicherte Snapshots

Kurz gesagt:

- der Browser öffnet über Hocuspocus eine dauerhafte WebSocket-Verbindung
- der Server lädt den letzten Snapshot aus der Datenbank
- beide Seiten synchronisieren danach laufend Änderungen in beide Richtungen
- dadurch entsteht ein kollaboratives Whiteboard in Echtzeit

Lasso-Selektion im Whiteboard

Die Lasso-Funktion im Whiteboard arbeitet nicht mit einem Screenshot oder einer Pixelmaske, sondern mit Geometrie auf den gespeicherten Board-Daten.

Einfach erklärt:

- während der Benutzer mit Maus oder Touch das Lasso zieht, werden fortlaufend Punkte gesammelt
- diese Punkte bilden ein Polygon, also eine geschlossene Fläche
- alle Strokes auf der aktuellen Seite liegen bereits als Punktlisten im Frontend vor
- beim Loslassen wird für jedes Element geprüft, ob es das Lasso-Polygon schneidet oder darin liegt
- alle passenden Elemente werden ausgewählt und können danach verschoben oder gelöscht werden

Woher kommen die Stroke-Daten? Die Auswahl liest die Strokes nicht direkt aus einer SQL-Abfrage.

Der Ablauf ist:

- das Board wird als Y.js-Dokument über den Collab-Service geladen
- der Collab-Service lädt dafür den letzten Snapshot aus `board_documents`
- im Frontend werden aus dem Y.js-Dokument die `elements` gelesen
- ein Stroke ist dabei ein Element mit `type: "stroke"` und einer Liste von Punkten

Vereinfacht sieht ein Stroke so aus:

```
{
  "id": "uuid",
  "type": "stroke",
  "pageId": "uuid",
  "color": "#111827",
  "size": 4,
  "points": [
    { "x": 0.18, "y": 0.27 },
    { "x": 0.19, "y": 0.28 },
    { "x": 0.21, "y": 0.29 }
  ]
}
```

Die Punkte sind normalisierte Board-Koordinaten:

- `x = 0` ist ganz links, `x = 1` ganz rechts
- `y = 0` ist ganz oben, `y = 1` ganz unten

Dadurch funktioniert dieselbe Logik unabhängig von Zoom, Displaygröße oder Auflösung.

Wie entsteht die Lasso-Fläche? Beim Start des Lassos wird der erste Pointer-Punkt gespeichert. Beim Bewegen kommen weitere Punkte hinzu. Diese Punkte werden nacheinander verbunden. Am Ende wird die Linie geschlossen.

Mathematisch ist das einfach ein Polygon:

`P = [p1, p2, p3, ..., pn]`

mit Punkten

`pi = (xi, yi)`

Die Kanten des Lassos sind dann:

`(p1,p2), (p2,p3), ..., (p(n-1),pn), (pn,p1)`

Die letzte Kante von `pn` zurück nach `p1` schließt die Fläche.

Wie wird ein Stroke gegen das Lasso geprüft? Ein Stroke besteht ebenfalls aus Punkten:

`S = [s1, s2, s3, ..., sm]`

und daraus ergeben sich Liniensegmente:

`(s1,s2), (s2,s3), ..., (s(m-1),sm)`

Ein Stroke gilt als ausgewählt, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

1. mindestens ein Stroke-Punkt liegt innerhalb des Lasso-Polygons
2. mindestens ein Stroke-Segment schneidet eine Kante des Lasso-Polygons

Praktisch bedeutet das:

- ein komplett innerhalb gezeichneter Stroke wird erkannt
- ein Stroke, der nur durch den Rand des Lassos läuft, wird ebenfalls erkannt
- ein Stroke muss also nicht vollständig innerhalb der Fläche liegen

Die Mathematik dahinter in einfacher Form Für die Punkt-in-Polygon-Prüfung wird das klassische Ray-Casting-Prinzip verwendet:

- man denkt sich von einem Punkt eine waagerechte Linie nach rechts
- man zählt, wie oft diese Linie den Polygonrand schneidet
- ungerade Anzahl von Schnitten: Punkt liegt innerhalb
- gerade Anzahl von Schnitten: Punkt liegt außerhalb

Für Segment-Schnittpunkte wird geprüft, ob sich:

- ein Segment des Strokes
- und eine Kante des Lassos

geometrisch kreuzen.

Damit wird die Auswahl robust, auch wenn:

- der Stroke nur teilweise im Lasso liegt
- die Linie das Lasso nur am Rand berührt
- das Lasso sehr frei und unregelmäßig gezeichnet wird

Warum ist dieser Ansatz sinnvoll? Der geometrische Ansatz hat mehrere Vorteile:

- keine teure Pixelanalyse notwendig
- unabhängig von Canvas-Auflösung und Zoom
- dieselben Daten können für Auswahl, Verschieben und Persistenz verwendet werden
- funktioniert gleich für lokale und kollaborative Whiteboard-Daten

Kurz gesagt:

- Y.js liefert die aktuellen Stroke-Punkte
- das Lasso erzeugt daraus ein Polygon
- die Anwendung prüft mathematisch Punkt-in-Polygon und Segment-Schnitt
- daraus ergibt sich die Auswahl

Datenbank

Die alte Datenbankbeschreibung im README wurde ausgelagert, weil das Modell inzwischen deutlich umfangreicher ist.

Die aktuelle Dokumentation findest du hier:

- DATABASE_AUFBAU.md
- images/liveclass_board_database.svg

Sie enthält:

- alle Tabellen
- Spalten
- Enums
- Foreign Keys
- wichtige Constraints
- Hinweise für UML / ERD

Zusätzlich gibt es eine visuelle Datenbankübersicht:

API-Überblick

Auth

- `POST /api/auth/login`
- `POST /api/auth/register`
- `POST /api/auth/guest`
- `POST /api/auth/refresh`
- `POST /api/auth/logout`
- `GET /api/auth/me`

Environments

- `GET /api/environments`
- `POST /api/environments`
- `POST /api/environments/join/:code`
- `PATCH /api/environments/:environmentId`
- `POST /api/environments/:environmentId/templates`
- `PATCH /api/environments/:environmentId/templates/:templateId`
- `PATCH /api/environments/:environmentId/members/:userId`
- `DELETE /api/environments/:environmentId`

Sessions

- `GET /api/sessions`
- `POST /api/sessions`
- `PATCH /api/sessions/:sessionId`
- `POST /api/sessions/:sessionId/boards`
- `PATCH /api/sessions/:sessionId/boards/:boardId`

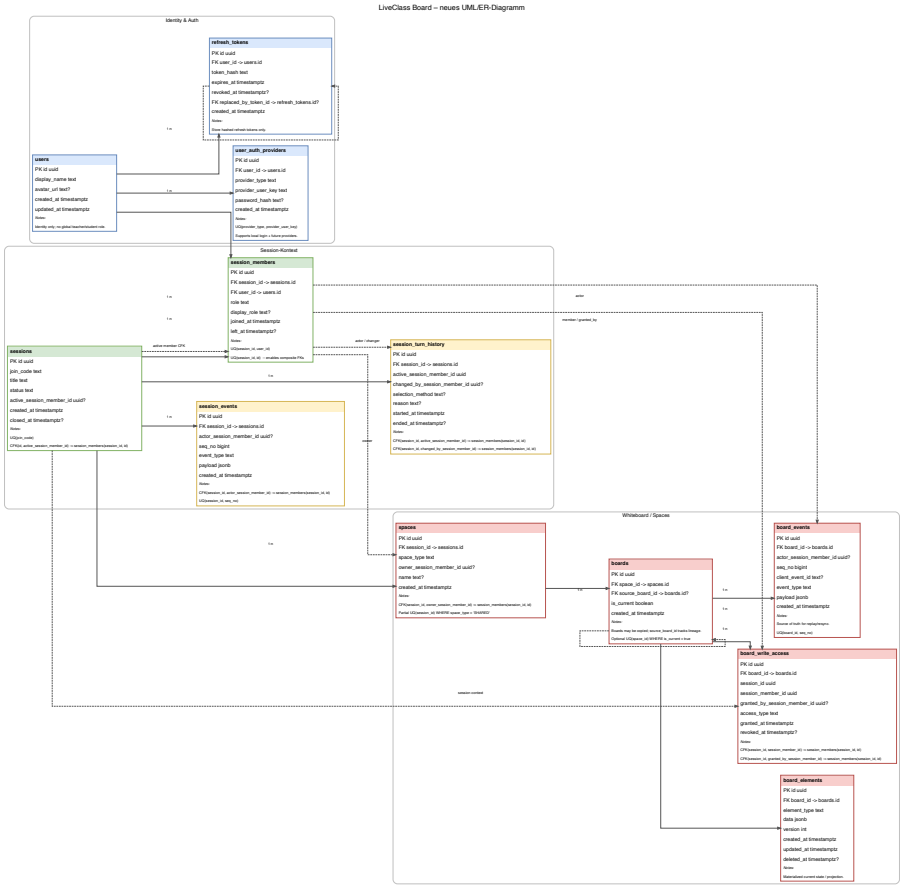

Figure 3: Datenbankübersicht

- DELETE /api/sessions/:sessionId/boards/:boardId
- GET /api/sessions/:code
- GET /api/sessions/:code/stream
- POST /api/sessions/:code/join
- POST /api/sessions/:sessionId/boards/:boardId/copy-personal
- POST /api/sessions/:sessionId/open
- POST /api/sessions/:sessionId/close
- POST /api/sessions/:sessionId/leave
- DELETE /api/sessions/:code
- POST /api/sessions/:sessionId/active-member

Frontend-Struktur

Die UI ist aktuell in drei Hauptseiten aufgeteilt:

```
apps/web/src/pages/
  LoginPage/
    LoginPage.jsx
    LoginPage.css
  DashboardPage/
    DashboardPage.jsx
    DashboardPage.css
  BoardPage/
    BoardPage.jsx
    BoardPage.css
```

Weitere wichtige Frontend-Dateien:

- `apps/web/src/App.jsx`: Routing und Silent-Login beim Start
- `apps/web/src/store/authStore.js`: Access Token und User-State
- `apps/web/src/lib/api.js`: API-Wrapper mit Auto-Refresh bei `401`
- `apps/web/src/hooks/useYjs.js`: Y.js-Integration für Pages, Elemente, Undo/Redo und Sync

Wichtige Tabellen

Der aktuelle Kern der Datenbank besteht aus:

- users
- user_auth_providers
- refresh_tokens
- environments
- environment_memberships
- environment_board_templates
- sessions
- session_members
- session_turn_history
- session_events

- `spaces`
- `boards`
- `board_elements`
- `board_write_access`
- `board_events`
- `board_documents`

Entwicklungsnotizen

- PostgreSQL ist die zentrale Persistenz für Fachdaten und Board-Snapshots.
- Redis wird für den verteilten Y.js-/Hocuspocus-Teil verwendet.
- Session-Realtime läuft über PostgreSQL `LISTEN/NOTIFY`.
- Der Compose-Stack startet zwei API-Instanzen hinter Traefik.
- Das Frontend wird als eigener Service gebaut und über Traefik ausgeliefert.