

Entwurfsdokumentation – StudiQ

1. Ziel der Entwurfsdokumentation

Ziel dieser Entwurfsdokumentation ist es, zukünftige Entwickler:innen bei der Weiterentwicklung und Wartung des Systems StudiQ zu unterstützen. Dazu beschreibt dieses Dokument:

- die wesentlichen architekturellen Anforderungen (Architekturtreiber),
- die Systemarchitektur auf logischer und technischer Ebene,
- zentrale technische Entscheidungen (Frameworks, Datenbank, Patterns),
- die wichtigsten Schnittstellen (REST-APIs),
- sowie die Zusammenarbeit zwischen den Komponenten anhand von Sequenzdiagrammen.

Die detaillierte API- und Klassenreferenz wird separat durch die automatisch generierte Doxygen-HTML-Dokumentation bereitgestellt.

2. Architekturtreiber und Anforderungen

2.1 Funktionale Anforderungen

- **Studierende** können Studiengänge, Module und Lernsets durchsuchen.
- Zu jedem Lernset können **Quizzes** gestartet, bearbeitet und erneut durchgeführt werden.
- Der individuelle Lernfortschritt (z. B. Anzahl gelöster Quizzes, richtige/falsche Antworten, Streak, IQ-Score/IQ-Punkte) wird persistent gespeichert.
- Ein **Modulux-Scraper** importiert Studiengänge und Module aus einem externen System.
- Eine **Suchfunktion** erlaubt das Auffinden von Lernsets, Quizzes, Modulen und Studiengängen anhand von Freitext.

Hinweis: Das „IQ-Level“ wird im Frontend aus dem IQ-Score abgeleitet und ist kein persistierter Backend-Wert.

2.2 Nichtfunktionale Anforderungen

Benutzbarkeit:

- Single-Page-Application mit flüssiger Navigation (Vue).
- Klare Hierarchie (Studiengang → Modul → Lernset → Quiz).

Wartbarkeit & Erweiterbarkeit:

- Trennung von Frontend und Backend.
- Im Backend saubere Trennung in Django-Apps (accounts, quizzes, scraper).
- Im Frontend Trennung in Views, Stores und Service-Schicht.

Testbarkeit:

- Klare REST-Endpunkte, die unabhängig vom Frontend getestet werden können.

Performanz:

- Pagination / Limitierung in der Suche.
- Berechnungen (z. B. Quiz-Logik, Bewertung, Fortschritt) komplett serverseitig.

Sicherheit:

- Authentifizierung per Session-basiertem Login (Django).
- Zugriff auf Quiz-Fortschritt nur für den angemeldeten User.

Diese Anforderungen sind die zentralen Architekturtreiber und haben direkt zu der getroffenen Technologie- und Strukturwahl geführt.

3. Systemkontext

3.1 Akteure

- **Student:in**
Hauptnutzer, bearbeitet Quizzes, betrachtet Lernfortschritt und Statistiken.
- **Administrator:in / Dozent:in (optional)**
Kann Quizzes/Lernsets pflegen (je nach Projektumfang).

3.2 Externe Systeme

- **Modulux**
Externe Quelle für Studiengänge und Module. Wird über den Scraper angebunden (in der Regel per HTTP-Requests / HTML-Scraping / API).
- **Datenbank (ausgelagert)**
Die relationale Datenbank (MySQL) ist als eigener Dienst betrieben (separat vom Django-Prozess) und wird vom Backend über ein DB-Interface angesprochen.

3.3 Kontextdiagramm (C4 Level 1 – verbal)

StudiQ besteht aus:

- **StudiQ Web-Client (Vue SPA):** Läuft im Browser, wird über HTTPS ausgeliefert.
- **StudiQ Backend (Django):** Stellt REST-Endpunkte bereit und implementiert die Geschäftslogik.
- **MySQL Datenbank (separater Dienst):** Persistiert Domänen- und Nutzerfortschritt.

Externe Verbindung: - Scraper Modulux (Pull von Modul- und Studiengangsdaten).

4. Architekturübersicht (C4 Level 2 – Container-Sicht)

Die Architektur folgt einem klassischen Client-Server-Ansatz.

Container:

- **Browser / Vue-Frontend:**
SPA, verantwortlich für Routing, Rendering der Views, lokales State-Management und Auslösen der API-Calls.
- **Django Backend / REST-API:**
Hostet die Apps accounts, quizzes und scraper. Bereitstellung von JSON-basierten REST-Endpunkten.

- **Datenbank (MySQL):**
Separater Dienst. Tabellen u. a. für User, Studiengänge, Module, Lernsets, Quizzes, Fragen, Antwortoptionen, QuizSessions, StudyDays, etc.
- **Modulux (extern):**
Wird nur vom scraper angesprochen. Liefert Domänendaten, die in die eigene Datenbank übernommen werden.

Die Kommunikation zwischen Frontend und Backend erfolgt ausschließlich über HTTPS/REST im JSON-Format.

5. Backend-Architektur (Django)

Das Backend ist in mehrere Apps untergliedert.

5.1 accounts-App

Verantwortung:

- Benutzerverwaltung (Registrierung, Login, Logout).
- User-Statistiken (z. B. IQ-Score/IQ-Punkte, solved quizzes, richtige/falsche Antworten).
- StudyDays zur Berechnung von Lern-Streaks.

Wichtige Klassen (Beispiele):

- **User:** Erweiterung des Django-Usermodells.
- **StudyDay:** Speichert, an welchen Tagen gelernt wurde (nicht die Lerndauer).

Views:

- LoginView, LogoutView, RegisterView
- MeView (liefert Daten des eingeloggten Users)
- UserStatsView, StudyCalendarView

Besondere Entscheidungen:

- Zentralisierte Statistiken pro User vermeiden redundante Berechnungen.
 - UserStatsView aggregiert Daten aus mehreren Tabellen.
 - Das Anlegen eines StudyDay aktualisiert die Streak automatisch (kein zusätzlicher expliziter Schritt erforderlich).
-

5.2 quizzes-App

Verantwortung:

- fachliche Domäne „Studiengang/Modul/Lernset/Quiz“.
- Verwaltung von Fragen, Antwortoptionen, Sessions und serverseitiger Quiz-Durchführung.

Wichtige Klassen:

- Domänenmodelle: Studiengang, Modul, Lernset, Quiz, Question, AnswerOption, QuizSession.
- ViewSets: u. a. StudiengangViewSet, ModulViewSet, LernsetViewSet, QuizViewSet.

Spezielle Views / Actions (aktuell):

- `QuizzesByLernsetView` – liefert alle Quizzes zu einem Lernset.
- `QuizViewSet` Actions (statt separater `CompletionView`):
 - `start` – erzeugt eine `QuizSession` und liefert die erste Frage.
 - `answer` – nimmt eine Antwort entgegen, bewertet serverseitig und liefert die nächste Frage / Feedback.
 - `complete` – beendet die Session (setzt `end_time`) und liefert Auswertung + persistierte Updates.
 - `sessions` – liefert Sessions zu einem Quiz (z. B. History).
- `LeaderboardViewSet` – Aggregation über mehrere User.
- `SearchView` – globale Suche über verschiedene Entitäten.
- `SuggestedQuizzesView` – Vorschläge basierend auf Fortschritt.

Serializers:

- Verschachtelte Serializer (z. B. `QuizSerializer` mit eingebetteten `QuestionSerializer` und `AnswerOptionSerializer`).
- „Short“-Serializer (z. B. `QuizForLernsetSerializer`) für Listenansichten zur Performanceoptimierung.

Wichtige Designentscheidungen (aktuell):

- Es gibt nur noch `QuizSession` als persistierten Fortschritts-/Durchführungscontainer.
- Eine Session wird über `.../start` erzeugt.
- `end_time` bleibt leer bis `.../complete` aufgerufen wird (dann wird sie auf den Aufrufzeitpunkt gesetzt).
- Die Session wird serverseitig beim Senden einer Antwort (`.../answer`) aktualisiert.
- Der Quiz-Ablauf ist serverseitig (Bewertung, Fortschritt, Auswertung), wodurch Manipulation/Inkonsistenzen reduziert werden und das Frontend schlanker bleibt.

5.3 scraper-App

Verantwortung:

- Import von Studiengängen und Modulen aus Modulux.

Designentscheidungen:

- Klar getrennt vom restlichen Code, um spätere Anpassungen an Modulux leicht zu ermöglichen.
 - Ergebnis wird in die gleichen Studiengang- und Modul-Modelle geschrieben.
-

6. Frontend-Architektur (Vue)

Das Frontend ist als SPA strukturiert und in Views, Router, Store und Services aufgeteilt.

6.1 Views (src/client/src/views)

Beispiele:

- **StudiengangView** – zeigt die Liste der Studiengänge.
- **ModulView** – zeigt ein Modul mit seinen Lernsets.
- **LernsetView** / **QuizOverviewView** – Quizzes zu einem Lernset.
- **QuizView** – UI für Quiz-Durchführung (Fragen anzeigen, Antworten wählen, Ergebnisse anzeigen).
- **QuizResultView** – Auswertung und Feedback.
- **SearchView** – globale Suche.

Jede View ist zuständig für: Darstellung, Auslösen der Service-Aufrufe, Aktualisierung des Stores.

6.2 Routing (src/client/src/router)

Beispiele:

- `/studiengaenge`
- `/studiengaenge/:id`
- `/module/:id`
- `/lernsets/:id`
- `/quiz/:id`
- `/quiz/:id/result`
- `/search`

Guard-Mechanismen (z. B. Zugriff nur für eingeloggte User) können über Navigation Guards implementiert werden.

6.3 Stores (src/client/src/stores)

- **userStore** – eingeloggter User (Name, Rolle, Statistiken inkl. IQ-Score).
- **appStore** – globale UI-Settings (Theme, Ladezustände etc.).
- Optional eigene Stores für z. B. Quiz-Editing.

6.4 API-Services (src/client/src/services)

Kapseln alle HTTP-Aufrufe zum Backend.

Beispiele:

- **auth.js** – Login/Logout, GET `/api/auth/me/`
- **user.js** – GET `/api/users/me/stats/`
- **quizzes.js** – GET `/api/quizzes/:id/`, POST `/api/quizzes/:id/start/`, POST `/api/quizzes/:id/answer`, POST `/api/quizzes/:id/complete/`, GET `/api/quizzes/:id/session/`
- **lernsets.js**, **modules.js**, **studiengaenge.js** – Laden der Hierarchie
- **search.js** – GET `/api/search/?q=...`

Designentscheidung: Durch die Service-Schicht hängt die UI nicht direkt am Client. Das erleichtert Tests und spätere API-Änderungen.

7. Schnittstellenbeschreibung (REST-API)

Die wichtigsten REST-Endpunkte werden in der Doxygen-Doku und im Code definiert. Hier ein Überblick zentraler Gruppen.

Auth & User

- POST /api/auth/login/ – Login, Rückgabe von User-Daten + Session.
- POST /api/auth/logout/ – Logout.
- GET /api/auth/me/ – Daten des eingeloggten Users.
- GET /api/users/me/stats/ – Statistiken (IQ-Score/IQ-Punkte, Streak, Ranking, etc.).

Studiengang / Modul / Lernset

- GET /api/studiengaenge/
- GET /api/modules/?studiengang=<id>
- GET /api/modules/:id/ – inkl. Liste der Lernsets.
- GET /api/lernsets/:id/ – Details zu einem Lernset.
- GET /api/lernsets/:id/quizzes/ – Quizzes zu einem Lernset.

Quiz (serverseitiger Ablauf)

- GET /api/quizzes/:id/ – Quiz-Metadaten, ggf. Fragen/Antwortoptionen je nach Serializer-Konzept.
- POST /api/quizzes/:id/start/ – Startet ein Quiz, erzeugt eine QuizSession und liefert die erste Frage.
- POST /api/quizzes/:id/answer – Sendet Antwort, serverseitige Bewertung + nächste Frage/Feedback.
- POST /api/quizzes/:id/complete/ – Beendet QuizSession (setzt end_time) und liefert Auswertung; aktualisiert Fortschritt & Statistiken.
- GET /api/quizzes/:id/sessions – Session-Historie (z. B. vergangene Durchläufe).

Suche

- GET /api/search/?q=<query>&limit=<n> – durchsucht Lernsets, Quizzes, Module, Studiengänge.

Alle Endpunkte liefern und akzeptieren JSON. Die konkrete Form der Payloads ist in den Serializern (serializers.py) definiert und durch Doxygen dokumentiert.

8. Sequenzdiagramme (Zusammenarbeit der Komponenten)

Die Zusammenarbeit zwischen Frontend und Backend wird anhand mehrerer Sequenzdiagramme modelliert und als Bilder in die Projektdokumentation integriert.

Login

Sequendiagramm Login

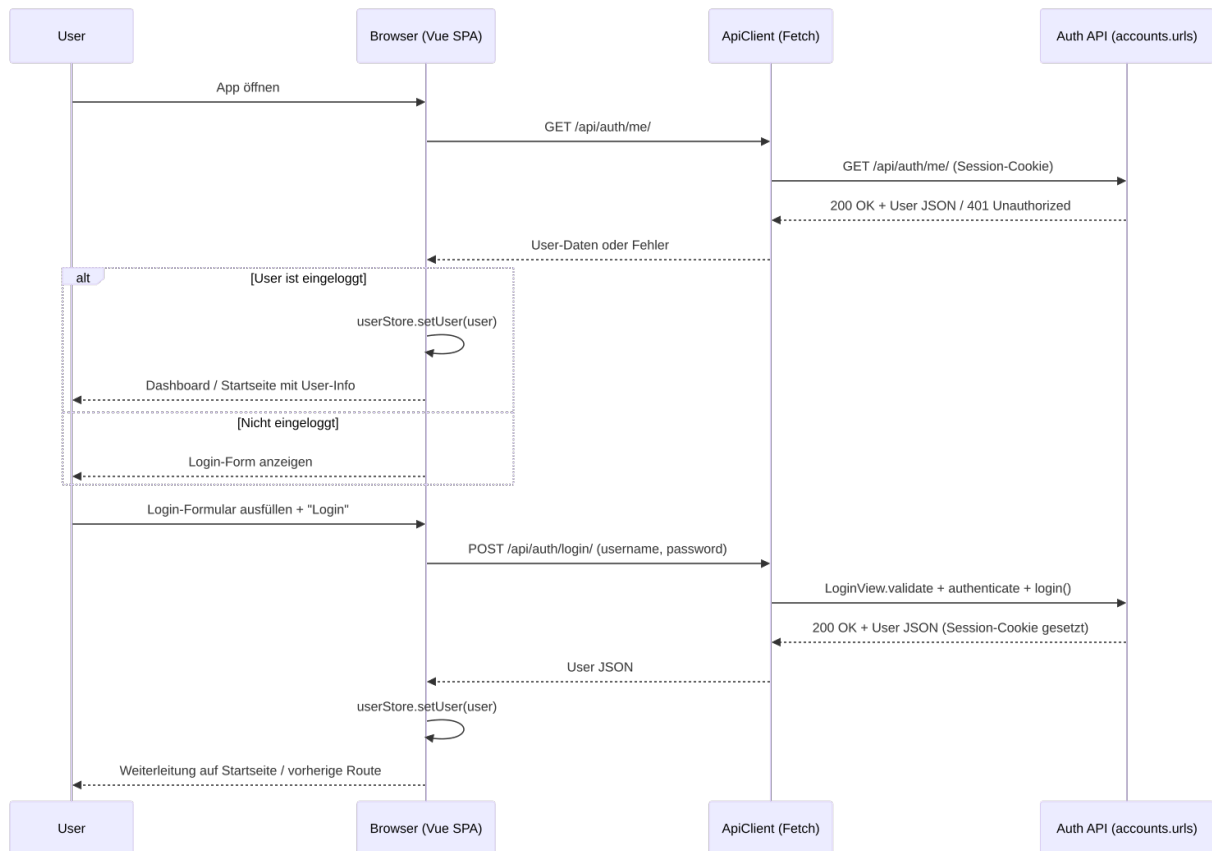


Figure 1: diagram

Quiz spielen

- Laden des Quizzes: `GET /api/quizzes/:id/`
- Starten des Quizzes und Holen der ersten Frage: `POST /api/quizzes/:id/start/`
- Senden der vom User ausgewählten Antwort + Bewertung + Empfang der nächsten Frage: `POST /api/quizzes/:id/answer`
- Abschluss (und Empfang der Auswertung) via `POST /api/quizzes/:id/complete/`
- Nachladen der Statistiken: `GET /api/users/me/stats/` ### Sequenzdiagramm

9. Wichtige technische Entscheidungen

Programmiersprachen & Frameworks

- **Backend:** Python, Django, Django REST Framework.
- **Frontend:** JavaScript/TypeScript mit Vue.
- **Dokumentation:** Doxygen für Backend + Frontend, Mermaid für Sequenzdiagramme.

Architekturmuster

- Trennung von Client/Server.
- RESTful API mit klaren Ressourcen (Studiengang, Modul, Lernset, Quiz, User).
- Im Frontend MVVM-artige Trennung: Views Store Services.

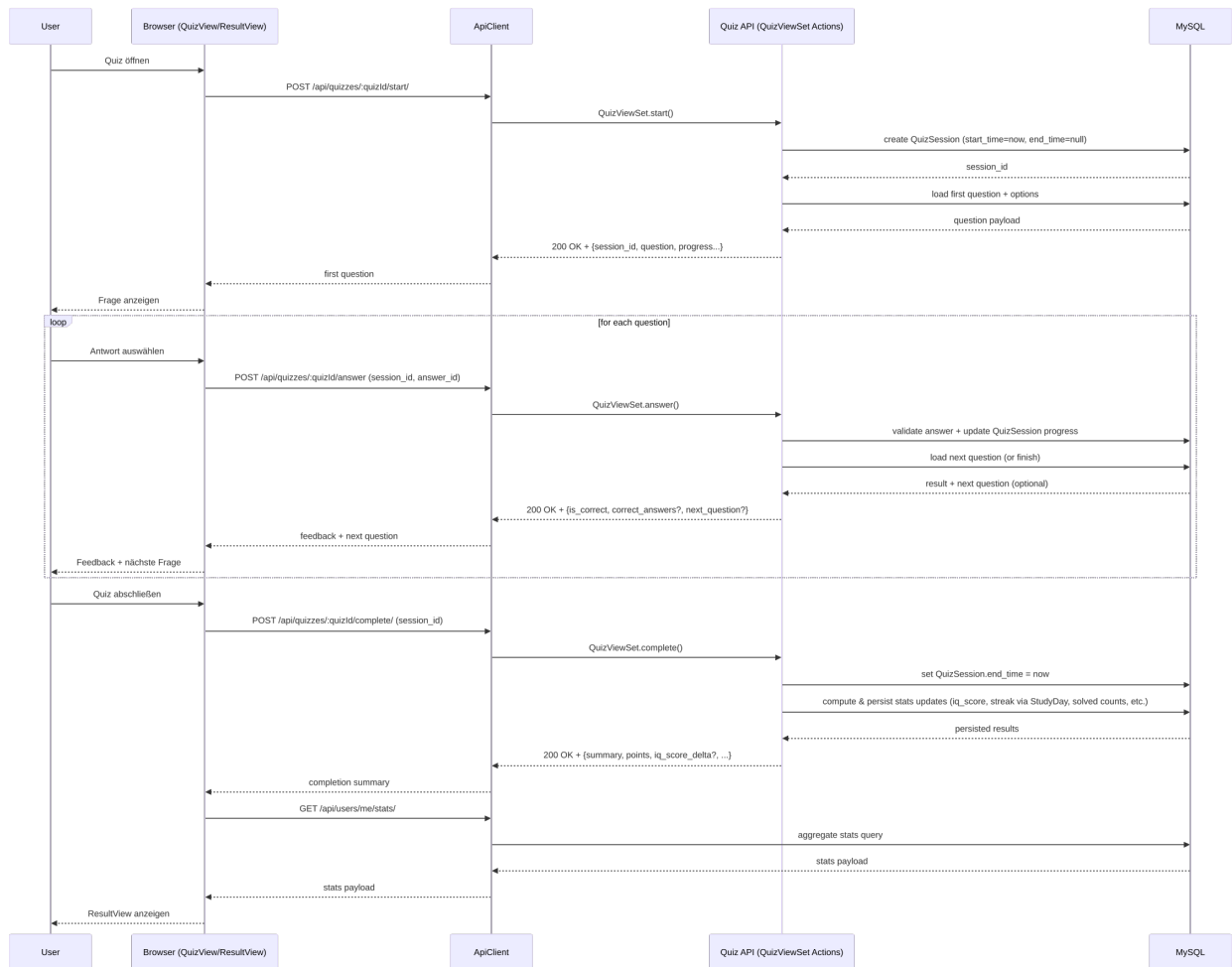


Figure 2: diagram

Persistenz

- Relationale Datenbank (MySQL) wegen strukturierter Domäne und klarer Relationen (FK-Beziehungen).
- DB als separater Dienst (ausgelagert) und über .env konfiguriert angebunden.

Konfigurierbarkeit

- Nutzung von .env (über python-dotenv) für DB-Credentials und sensitive Settings.

Dokumentation

- Automatische Generierung der Klassen-/Modulreferenz mit Doxygen.
 - Manuelle Entwurfsdokumentation (dieses Dokument)
-

10. Konsistenz der Sichten

Die verschiedenen Sichten auf das System – Anforderungen, Architekturdiagramme (C4), Sequenzdiagramme, REST-Schnittstellen und Code-Dokumentation (Doxygen) – sind aufeinander abgestimmt:

- Alle in den Diagrammen erwähnten Komponenten existieren im Code (ViewSets/Actions, Serializer, Models, Services).
- Die beschriebenen Endpunkte entsprechen der tatsächlichen URL-Konfiguration im Django-Projekt.
- Die fachliche Hierarchie Studiengang → Modul → Lernset → Quiz wird sowohl im Datenmodell, in den API-Responses als auch in der UI-Navigation konsistent dargestellt.