

LB-Projekt Modul 223

Natalia & Anina

Abgabedatum: 31.01.2026

Lehrer: Graziano Laveder

Inhalt

Unsere Webapplikation	2
Vorgaben.....	2
Was soll unsere App sein?	2
Anforderungen (funktional & nicht-funktional).....	3
Funktionale Anforderungen.....	3
Nicht-funktionale Anforderungen.....	3
Gesamtarchitektur (Übersicht)	3
Backend-Dokumentation	3
Gesamtidee der Architektur	4
ER-Diagramm.....	5
JWT-Authentifizierungs-Flow	5
Technologie-Stack Backend	6
Frontend-Dokumentation	7
AuthContext.....	8
Token-Handling	8
Login / Logout Flow.....	8
API-Anbindung	9
Axios	9
Vite.....	9
React Router	9
Technologie-Stack Frontend.....	9
Multiuser-Konzept	10
Konfiguration:	10
Testing mit Swagger:	12
Test 1	12
Reflexion:	14
Installationsanleitung:	14
1. Voraussetzungen.....	14
2. Deployment (Docker).....	14
Hilfestellung:	15

Unsere Webapplikation

Vorgaben

Die Anwendung besteht aus zwei Hauptkomponenten: einem **Frontend in Form einer Webapplikation** sowie einem **Backend mit persistenter Datenbank**.

Das Frontend dient der Benutzerinteraktion, während das Backend die Geschäftslogik, Authentifizierung (JWT) und den Datenzugriff übernimmt.

Die Anwendung ist **mehrbenutzerfähig (Multiuser)** und wurde eine **tokenbasierte Authentifizierung (z. B. JWT)** einsetzen.

Beide Komponenten werden **containerisiert mit Docker** betrieben und über eine gemeinsame **docker-compose-Konfiguration** orchestriert.

Was soll unsere App sein?

Das Projekt *GlobeNotes* ist eine Webapplikation, welche Reisenden ermöglicht, persönliche Reiseziele zu verwalten.

Der Benutzer kann:

- Reiseziele erfassen, anzeigen und löschen
- Kategorien & Reisearten filtern
- Bilder speichern
- Daten bestehen auch nach Neustart dank persistenter Datenhaltung

Technisch wurde das System so umgesetzt,

dass es vollständig **containerisiert** betrieben und **automatisiert deployt** werden kann. Die Applikation implementiert eine **tokenbasierte Authentifizierung (z. B. JWT)**, um geschützte Bereiche abzusichern und Benutzer eindeutig zu identifizieren.

Zusätzlich ist ein **Rollen- und Berechtigungskonzept** umgesetzt, bei dem zwischen den Rollen **Admin** und **User** unterschieden wird. Der Zugriff auf Funktionen und Endpunkte ist abhängig von der jeweiligen Rolle eingeschränkt.

Admin kann zusätzlich:

- Reiseziele verwalten
- Reisen löschen

Der technische Fokus liegt bewusst auf:

- Frontend–Backend-Kommunikation über eine REST-Schnittstelle
- Deployment über Docker Compose
- CI/CD-Pipeline in GitLab (vom Modul 210)
- Datenbank mit Docker

Anforderungen (funktional & nicht-funktional)

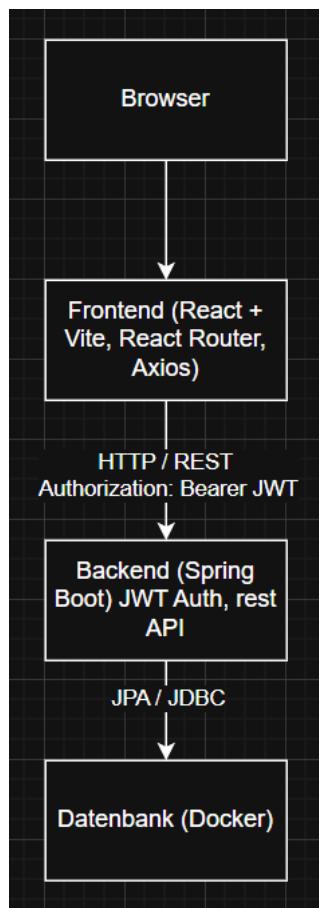
Funktionale Anforderungen

- Benutzer registrieren
- Benutzer einloggen (JWT)
- Reiseziele öffnen/erstellen
- Reiseziele speichern
- Rollen (User / Admin)

Nicht-funktionale Anforderungen

- Sicherheit (JWT, BCrypt)
- Mehrbenutzerfähig
- Datenpersistenz
- Trennung Frontend / Backend
- Wartbarkeit

Gesamtarchitektur (Übersicht)



Das System besteht aus einem React-Frontend, welches über HTTP/REST mit einem Spring-Boot-Backend kommuniziert. Die Authentifizierung erfolgt über JWT, wobei User-ID und Rolle im Token enthalten sind. Das Backend greift über JPA auf eine containerisierte Docker Datenbank zu.

Backend-Dokumentation

Zu Beginn des Projekts wurde die Applikation mit einer lokalen H2-Datenbank betrieben. Diese Variante eignete sich gut für die frühe Entwicklungsphase, da sie ohne zusätzliche Installation auskommt und eine schnelle Inbetriebnahme des Backends ermöglicht.

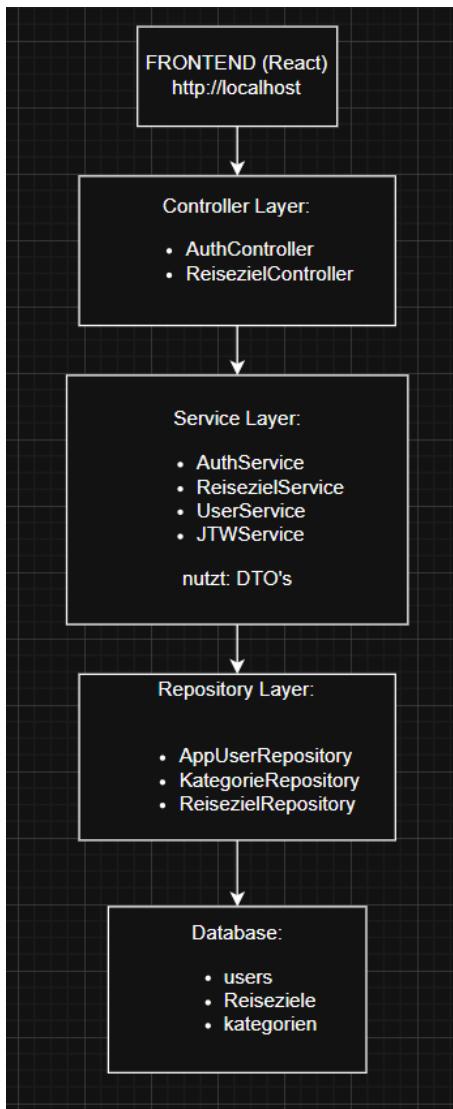
Im Verlauf des Projekts zeigte sich jedoch, dass diese Lösung für den geplanten Einsatz im Rahmen eines containerisierten Deployments nicht optimal ist.

Aus diesem Grund wurde entschieden, die Datenbank vollständig in das Docker-Setup zu integrieren. Die H2-Datenbank wird nun innerhalb eines Containers betrieben und über ein persistentes Docker-Volume gespeichert. Dadurch bleiben die Daten auch nach einem Neustart der Container erhalten und das gesamte System kann über Docker Compose einheitlich gestartet und betrieben werden.

Diese Umstellung ermöglicht ein konsistentes Deployment, verbessert die Wartbarkeit der Applikation und entspricht den Anforderungen an eine moderne, containerbasierte Systemarchitektur.

Gesamtidee der Architektur

Das Backend ist in klar getrennte Schichten aufgebaut. Jede Schicht hat eine klar definierte Aufgabe, was Wartbarkeit, Testbarkeit und Erweiterbarkeit verbessert.



Frontend → Controller

Das React-Frontend kommuniziert über HTTP mit dem Backend. Die Controller stellen REST-Endpunkte zur Verfügung und nehmen Requests entgegen, enthalten aber keine Business-Logik.

Controller Layer

Die Controller sind ausschliesslich für die Entgegennahme und Validierung von Requests zuständig. Sie delegieren die eigentliche Verarbeitung an die Service-Schicht.

Service Layer

Die Service-Schicht enthält die zentrale Geschäftslogik der Applikation. Hier werden Benutzer authentifiziert, Reiseziele verarbeitet und Zugriffsregeln umgesetzt. Zur Entkopplung von API und Datenbank werden DTOs verwendet.

Hier kannst du zusätzlich sagen:

- `JWTService` → Token erzeugen & prüfen
- `UserService` → User + Rollen
- `ReisezielService` → Multiuser-Logik

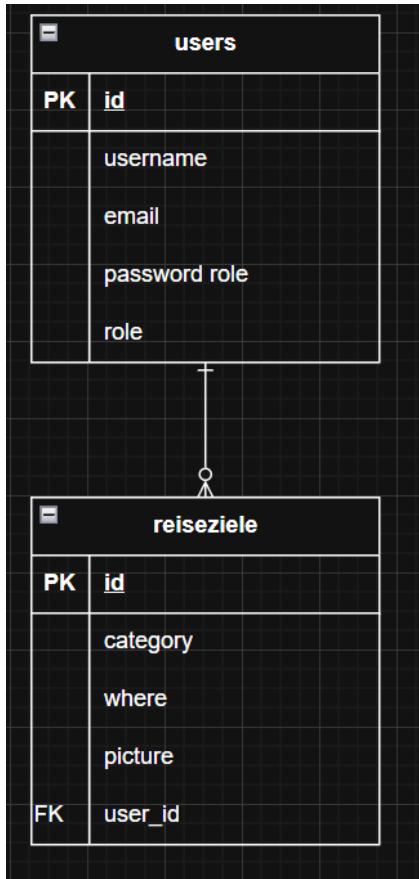
Repository Layer

Die Repository-Schicht kapselt den Datenbankzugriff. Sie verwendet Spring Data JPA und ermöglicht den Zugriff auf persistente Daten ohne direkte SQL-Abfragen im Code.

Datenbank

Die Datenbank speichert Benutzer, Reiseziele und Kategorien. Der Zugriff erfolgt ausschliesslich über die Repository-Schicht, um eine saubere Trennung der Verantwortlichkeiten sicherzustellen.

ER-Diagramm



Das ER-Diagramm zeigt die relationale Datenstruktur der Applikation.

Zwischen den Entitäten *users* und *reiseziele* besteht eine **one to many** Beziehung.

Ein Benutzer kann mehrere Reiseziele besitzen, während jedes Reiseziel genau einem Benutzer zugeordnet ist.

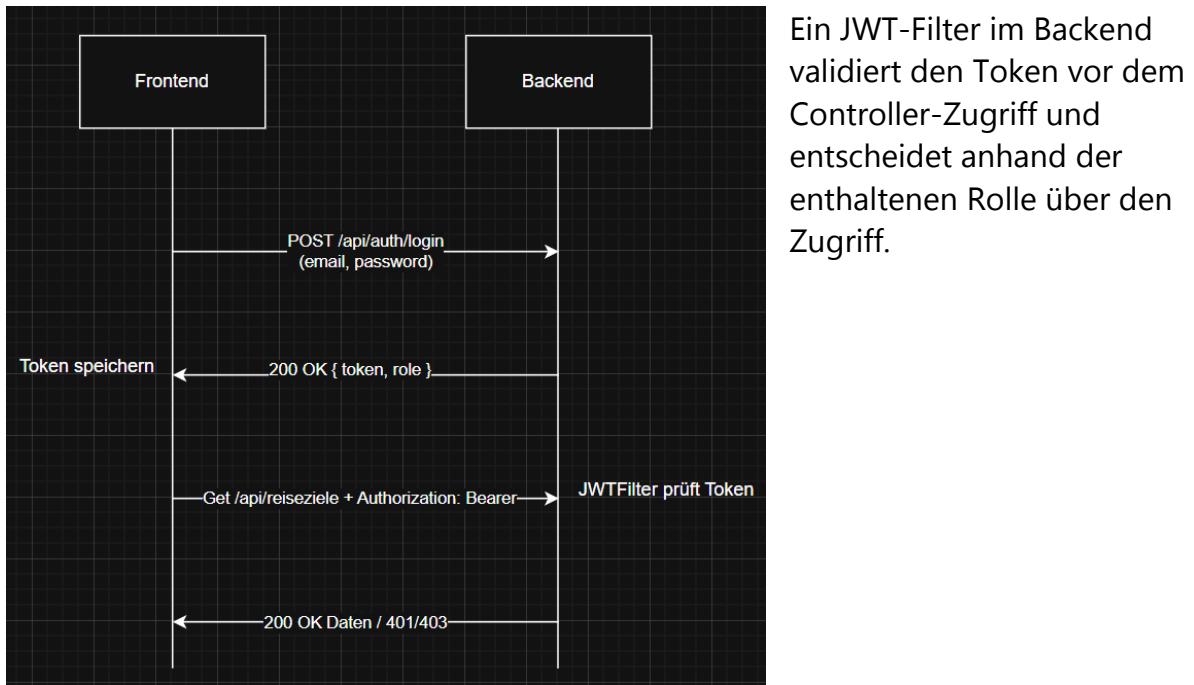
Die Zuordnung erfolgt über den Fremdschlüssel *user_id* in der Tabelle *reiseziele*.

Diese Struktur stellt sicher, dass Reiseziele eindeutig einem Benutzer zugewiesen sind und die Multiuser-Funktionalität korrekt umgesetzt ist.

JWT-Authentifizierungs-Flow

Beim Login sendet das Frontend die Zugangsdaten an das Backend. Nach erfolgreicher Prüfung wird ein JWT zurückgegeben, welches im Frontend gespeichert wird.

Bei allen weiteren geschützten API-Aufrufen wird der Token im Authorization-Header als Bearer Token mitgesendet.



Technologie-Stack Backend

Technologie	Version	Verwendung
Java	21.0.9	Für Backend Sprache
Spring Boot	3.5.3	Backend-Framework
Spring Security	6.x	Authentifizierung & Autorisation
JWT	0.11.5	Token-basierte Auth
JPA	3.5.3	ORM für Datenbank
Docker	29.1.3	Datenbank
Maven	3.9.11	Build-Tool & Dependency Management

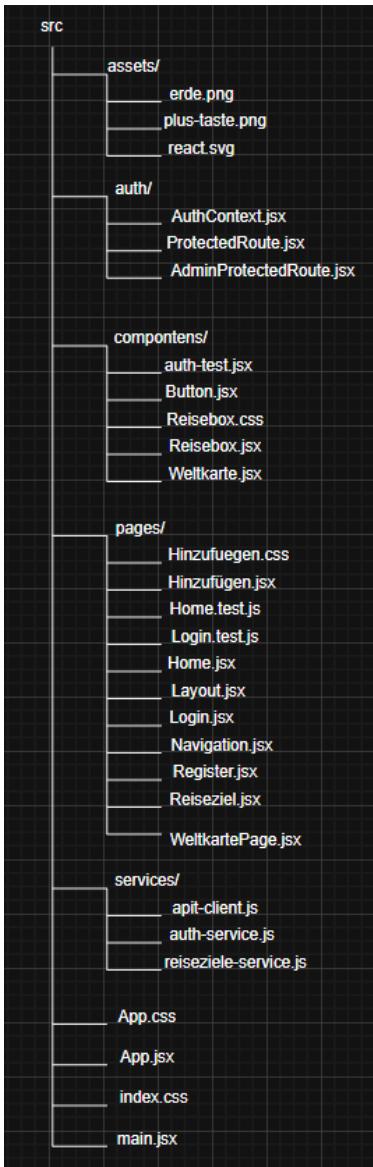
Beschreibung:

Das Backend basiert auf **Java 21** und **Spring Boot** und stellt eine REST-API bereit. Die Absicherung erfolgt über **Spring Security** mit **JWT-basierter Authentifizierung**.

Der Datenzugriff wird mit **JPA** umgesetzt, während **Docker** ein containerisiertes Deployment ermöglicht.

Der Build-Prozess wird über **Maven** gesteuert.

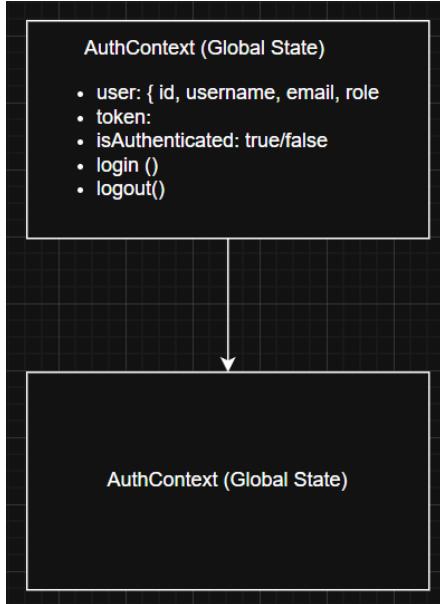
Frontend-Dokumentation



Das Frontend ist klar und modular aufgebaut. Assets enthalten statische Ressourcen wie Icons und Bilder. Der Ordner auth kapselt die komplette Authentifizierungslogik inklusive Context, geschützter Routen und Admin-Zugriff. Components beinhalten wiederverwendbare UI-Bausteine, während pages die einzelnen Seiten der Anwendung (z. B. Login, Home, Hinzufügen, Weltkarte) abbilden. Die services sind für die API-Kommunikation zuständig und trennen Backend-Zugriffe sauber von der UI-Logik. Diese Struktur sorgt für gute Wartbarkeit, klare Verantwortlichkeiten und einfache Erweiterbarkeit.

AuthContext

Im Frontend wird ein zentraler AuthContext verwendet, um Authentifizierungsdaten (z. B. Token, Username, Login-Status) global verfügbar zu machen. Dadurch müssen Komponenten den Login-Status nicht über Props “durchreichen” (Prop-Drilling), sondern können ihn direkt aus dem Context lesen.



Vorteile:

- Einheitliche Auth-Logik an einem Ort
- Saubere Trennung: UI-Komponenten bleiben “dumm”, Auth-Logik liegt zentral
- Einfacher Zugriff in jeder Route/Komponente

Token-Handling

Das Token wird nach erfolgreichem Login gespeichert, damit die Session beim Reload erhalten bleibt.

Ablauf:

1. Login-Response liefert JWT (Token)
2. Token wird:
 - im Context-State gehalten (für direkten Zugriff)
 - zusätzlich persistent gespeichert (meist localStorage oder sessionStorage)
3. Beim App-Start wird geprüft:
 - existiert ein Token im Storage?
 - dann setze AuthContext automatisch auf „eingeloggt“

Login / Logout Flow

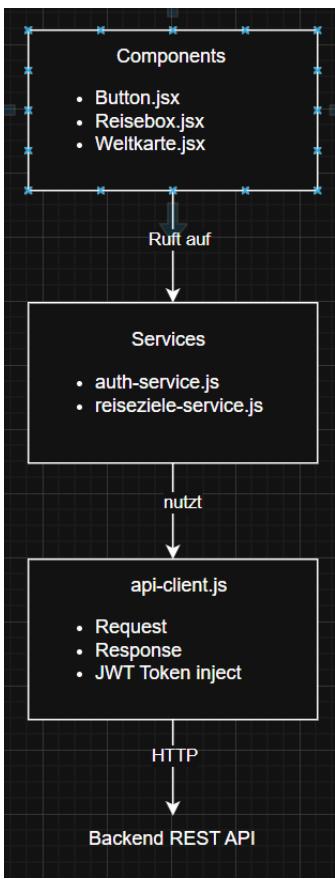
Login:

1. User gibt (Username/Email + Passwort) ein
2. Frontend sendet Request an Backend-Endpunkt /api/auth/login
3. Backend prüft Daten und liefert Token zurück
4. Frontend speichert Token + aktualisiert AuthContext
5. User wird weitergeleitet Home

Logout:

1. Token aus Context entfernen
2. Token aus Storage löschen
3. Man sieht nur noch die Login page

API-Anbindung



Services

Für Backend-Kommunikation werden Service-Dateien genutzt. Ziel:

- Komponenten enthalten keine Axios-Requests direkt
- API-Aufrufe werden gekapselt (besser testbar, übersichtlicher)

Axios

Als HTTP-Client wird Axios verwendet, weil:

- Requests/Responses einfacher zu handhaben sind
- Interceptors möglich sind
- Errors einheitlich verarbeitet werden können

Vite

Vite dient als Build-Tool/Dev-Server:

- Sehr schnelle Startzeit
- Schnelles Hot-Reloading
- Moderne Standard-Konfiguration für React-Projekte

React Router

Für die Navigation wird React Router eingesetzt:

- Seiten/Views werden über Routes abgebildet (`/login`, `/home`, `/reiseziel`, ...)
- Private Routes möglich (nur erreichbar wenn eingeloggt)
- sauberes Routing statt “alles in einer Seite”

Technologie-Stack Frontend

Technologie	Version	Verwendung
React	18.3.1	UI-Framework
Vite	5.3.4	Build-Tool & Dev-Server
React Router	6.26.1	Client-Side Routing
Context API	React 18	Globales State-Management (Auth)
React Testing Library	16.0.1	Component Testing

Beschreibung:

Das Frontend wurde mit React umgesetzt und bietet eine komponentenbasierte Benutzeroberfläche. Vite dient als modernes Build-Tool und Entwicklungsserver mit schnellen Ladezeiten. Für die Navigation innerhalb der Anwendung wird React Router verwendet. Das globale State-Management für die Authentifizierung erfolgt über die React Context API. Die Qualität und Funktionalität einzelner Komponenten wird mit der React Testing Library überprüft.

Multiuser-Konzept

Wir haben ein **Rollen- und Benutzerkonzept** mit **Admin** und **User** umgesetzt.

Die Unterscheidung erfolgt über die **Rolle im JWT**, welches beim Login erzeugt wird.

Aus dem JWT wird die **User-ID** gelesen und im Backend verwendet, um **user-spezifische Daten** zu laden oder zu speichern.

Bestimmte Funktionen und Seiten sind **rollenabhängig geschützt**:

Admins haben erweiterte Rechte, während normale User nur ihre eigenen Daten sehen können.

Konfiguration:

In diesem Kapitel werden die wichtigsten Konfigurationen beschrieben, welche für den Betrieb der Applikation notwendig sind.

Dazu gehören Docker Compose, Backend- und Frontend-Konfiguration sowie die CI/CD-Pipeline.

Docker-basierte Konfiguration

Die Applikation wird vollständig containerisiert betrieben. Die Koordination der einzelnen Komponenten erfolgt über **Docker Compose**.

Alle Konfigurationsdateien befinden sich im Ordner infra/ und ermöglichen ein direktes Deployment ohne manuelle Nacharbeiten.

Docker Compose Konfiguration

Die zentrale Datei für das Deployment ist die docker-compose.yml.

Sie definiert alle Services, Volumes und Ports, die für den Betrieb der Applikation notwendig sind.

Enthaltene Services

Service	Beschreibung
globenotes-backend	Spring Boot Backend inkl. REST-API
globenotes-frontend	React Frontend, ausgeliefert über NGINX
docker	Persistente Speicherung mit Docker
uploads (Volume)	Speicherung von hochgeladenen Bildern

Ports

Komponente	Port
Frontend	80
Backend	8080

Backend Konfiguration (Spring Boot)

Das Backend basiert auf **Spring Boot** und wird als ausführbares JAR innerhalb eines Docker-Containers betrieben.

Frontend Konfiguration (React + NGINX)

Das Frontend basiert auf **React (Vite)** und wird in einem **Multi-Stage-Dockerfile** gebaut.

API-Anbindung

Das Frontend kommuniziert über HTTP mit dem Backend:

`http://localhost:8080`

Die API-URL ist im Code zentral definiert, sodass keine Anpassung beim Deployment notwendig ist.

Testing mit Swagger:

Test 1

Mit diesem Test wird geprüft, ob die Registrierung eines neuen Benutzers korrekt funktioniert und ob das Backend nach erfolgreicher Registrierung direkt einen gültigen JWT-Token zurückgibt. Dadurch ist sichergestellt, dass ein neuer User ohne zusätzliche Schritte authentifiziert werden kann und die Antwort die nötigen User-Daten enthält (userId, username, role).

The screenshot displays two separate API test sessions in the Swagger UI.

Test 1: auth-controller POST /api/auth/register

Request Body:

```
{ "username": "testUser", "email": "testUser@test.ch", "password": "test123" }
```

Responses:

200 Response body (application/json)

```
{"token": "eyJhbGciOiJIUzI1NiJ9.yz7edt01x1e68Nce0b0010121c2WqM11x2b60511d081jw4vT9yvHf1y.C3m6Q0038jwQ2n39.TDw0B34elgCBHkAkyM6r1ZerAyqfslA1vY", "userId": 1, "username": "testUser", "role": "USER"}
```

Test 2: auth-controller POST /api/auth/login

Request Body:

```
{ "email": "testUser@test.ch", "password": "test123" }
```

Responses:

200 Response body (application/json)

```
{"token": "eyJhbGciOiJIUzI1NiJ9.yz7edt01x1e68Nce0b0010121c2WqM11x2b60511d081jw4vT9yvHf1y.C3m6Q0038jwQ2n39.TDw0B34elgCBHkAkyM6r1ZerAyqfslA1vY", "userId": 1, "username": "testUser", "role": "USER"}
```

Test 2

Dieser Test stellt sicher, dass geschützte Endpoints nicht ohne gültigen JWT-Token aufgerufen werden können. Ein Request ohne Authorization-Header muss mit 401/403 beantwortet werden. Damit ist die grundlegende Sicherheitsanforderung (Authentication erforderlich) automatisiert abgesichert.

The screenshot shows the 'reiseziel-controller' API test interface. A 'GET /api/reiseziele' request is selected. The 'Responses' section displays a 403 error response with the following details:

```
curl -X 'GET' \
  'http://localhost:8080/api/reiseziele' \
  -H 'Accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5c2lkIjoiMDA2LjZlYmQwMjIyNjIwOTk1IiwidHlwZSI6Imh0dHA6Ly9leGFtcGxlLmNvbS8xMDAxLjEwLjEwLjIwMCIsImlhdCI6MTY4ODUyNjUyMSwiZXhwIjoxNjg4ODUyNjUyLCJzY29wZW50IjoxfQ=='
Request URL
http://localhost:8080/api/reiseziele
Server response
Code Details
403 Error: response status is 403
Response headers
cache-control: no-cache,no-store,max-age=0,must-revalidate
connection: keep-alive
content-length: 0
date: Sat, 24 Mar 2028 13:27:47 GMT
expires: 0
keep-alive: timeout=60
pragma: no-cache
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-content-type-options: nosniff
x-xss-protection: 0
```

Test 3

Dieser Test prüft, ob das Login-Formular nach erfolgreichem Login den Token korrekt verarbeitet. Erwartet wird, dass der Token im Storage (z. B. localStorage) abgelegt wird und der Auth-State gesetzt wird. Damit ist bestätigt, dass das Frontend die Grundlage für geschützte Requests korrekt vorbereitet.

The screenshot shows the 'reiseziel-controller' API test interface. A 'GET /api/reiseziele' request is selected. The 'Responses' section displays a 200 success response with the following details:

```
curl -X 'GET' \
  'http://localhost:8080/api/reiseziele' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5c2lkIjoiMDA2LjZlYmQwMjIyNjIwOTk1IiwidHlwZSI6Imh0dHA6Ly9leGFtcGxlLmNvbS8xMDAxLjEwLjEwLjIwMCIsImlhdCI6MTY4ODUyNjUyMSwiZXhwIjoxNjg4ODUyNjUyLCJzY29wZW50IjoxfQ=='
Request URL
http://localhost:8080/api/reiseziele
Server response
Code Details
200 Response body
[{"id": 1, "name": "Berlin", "lat": 52.5200, "lon": 13.4050}
]
Response headers
cache-control: no-cache,no-store,max-age=0,must-revalidate
content-length: 103
content-type: application/json
date: Sat, 24 Mar 2028 13:27:13 GMT
expires: 0
keep-alive: timeout=60
 pragma: no-cache
transfer-encoding: chunked
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-content-type-options: nosniff
x-xss-protection: 0
```

The screenshot shows the H2 Console interface. On the left, there is a tree view of the database schema with tables like APP_USERS, REISEZIEL, and KATEGORIE. In the main area, a SQL statement is entered: "SELECT * FROM REISEZIEL;". Below it, the result of the query is displayed in a table:

ID	BILD_PFAF	BILD_URL	HIGHLIGHTS	JAHR	LATITUDE	LONGITUDE	ORT	KATEGORIE_ID	OWNER_ID
1	/uploads/023668fe-ca60-4f39-b8a5-889ca11eb22f_Screenshot 2025-06-06 094343.png	null	NewYear	2025	39.3260685	-4.8379791	Spanien	1	1

(1 row, 2 ms)

There is also an "Edit" button below the table.

Reflexion:

Während der Umsetzung des Projekts haben wir zunächst mit einer **H2-Datenbank** gearbeitet, die wir im vorherigem Modul eingebaut hatten für die einfache Datenspeicherung. Im Verlauf des Projekts haben wir jedoch gemerkt, dass H2 für unser **Multiuser- und Docker-Setup** nicht optimal ist. Deshalb haben wir uns entschieden, auf eine **Datenbank in Docker** umzusteigen, um eine realistischere und stabilere Umgebung zu haben.

Der Umstieg war lehrreich, aber auch herausfordernd. Wir hatten mehrere **Probleme mit Versionen**, insbesondere bei **Java, Spring Boot, Docker und Abhängigkeiten**, was zu Build-Fehlern und Laufzeitproblemen geführt hat. Zusätzlich mussten Konfigurationen wie Ports, Umgebungsvariablen und Security-Einstellungen angepasst werden.

Wir mussten jeden Abend zusammensitzen und an diesem Projekt arbeiten, da es immer wieder zu Problemen kamen, welche wir wieder fixen mussten.

Installationsanleitung:

1. Voraussetzungen

- Docker Desktop installiert
- GitHub Repo geklont

2. Deployment (Docker)

Docker desktop starten
cd infra
docker compose up --build

Danach erreichbar via:

<http://localhost> → Frontend

<http://localhost:8080> → Backend

Logindaten:

Benutzername: User

Passwort: User123

Benutzername: Admin

Passwort: Admin123

Hilfestellung:

- Austausch mit Mitschülern
- Unterstützung durch Online-Ressourcen (z. B. Forenbeiträge, Anleitungen zu Docker Compose, Kursmaterialien inkl. Websites)
- ChatGPT wurde gezielt zur Erklärung von Docker- und CI/CD-Konzepten. verwendet.