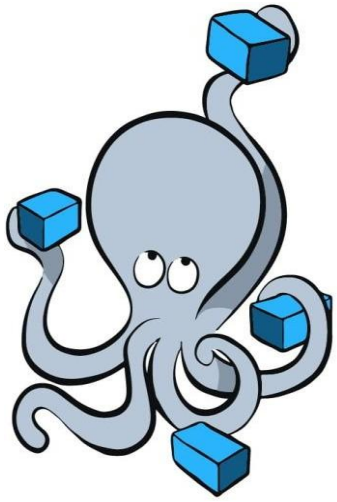


docker
Compose

LB-Projekt Modul 210

Natalia C Anina

Abgabedatum: 11.12.2025

Lehrer: Patrick Venzin

Projektverlauf

Diese Dokumentation beschreibt sowohl den Projektverlauf als auch den finalen Endzustand der Applikation.
Frühere Projektideen und Infrastrukturen werden dokumentiert, um den Entscheidungsprozess nachvollziehbar darzustellen.

Erste Projektidee (verworfenes Konzept):

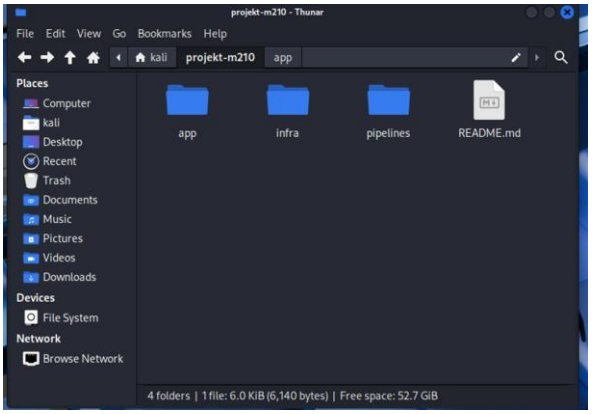
Zu Beginn des Projekts war geplant, eine einfache Webapplikation in Form eines Rezepteblogs umzusetzen.
Die Anwendung sollte auf WordPress basieren und über eine MariaDB-Datenbank verfügen.

Die Infrastruktur war so vorgesehen, dass WordPress und MariaDB jeweils in separaten Docker-Containern betrieben und über eine gemeinsame docker-compose.yml verwaltet werden. Zusätzlich sollte eine GitLab-CI/CD-Pipeline eingesetzt werden, um die Konfiguration zu validieren und ein automatisiertes Deployment zu ermöglichen.

Die geplante Anwendung sollte es Benutzern erlauben, Rezepte mit Titel, Beschreibung, Zutaten und optional Bildern zu erfassen und über einen Webbrowser zu verwalten. Die Nutzung wäre über das WordPress-Backend erfolgt.

Diese Projektidee diente als erste Annäherung an die Aufgabenstellung, wurde jedoch im weiteren Projektverlauf zugunsten eines anderen Ansatzes verworfen.

Erste GIT Repository s Grundstruktur:

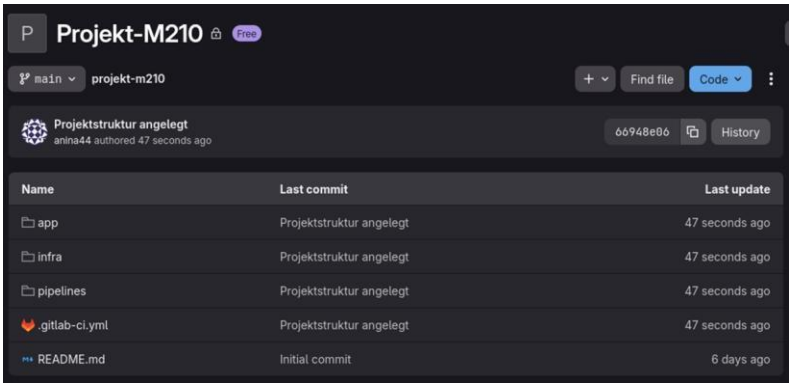


Wir wollten es bereits auf Gitlab commiten aber es war noch nicht möglich, da momentan nur die Ordnerstruktur festgelegt ist und noch keine Files vorhanden sind.

Files angelegt:

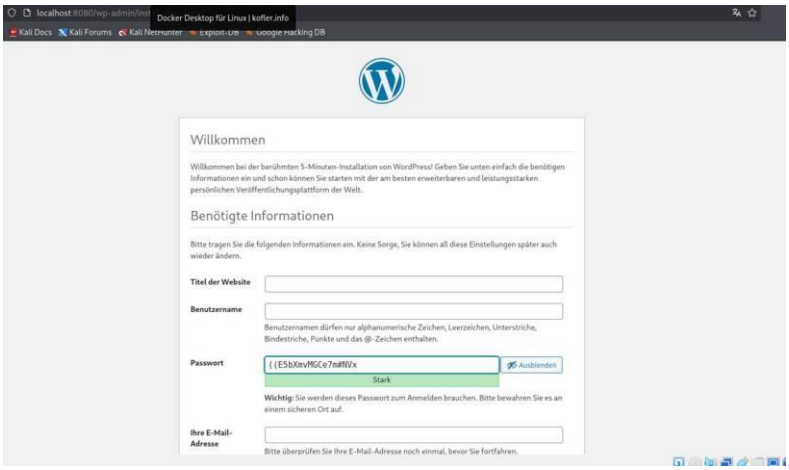
```
└─ Projekt/
   ├─ .gitlab-ci.yml
   │
   ├─ infra/
   │   ├─ docker-compose.yml
   │   ├─ .env.example
   │   └─ .gitkeep
   │
   ├─ app/
   │   └─ .gitkeep
   │
   ├─ pipelines/
   │   ├─ DEPLOYMENT-NOTIZEN.md   (optional)
   │   └─ .gitkeep
   │
   └─ README.md     (optional: erklärt Zweck der Ordner)
```

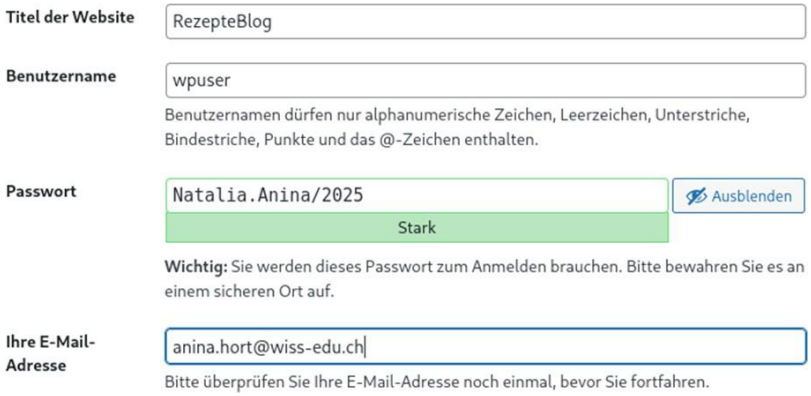
Nun konnten wir es ins GitLab commiten:
git add .
git commit -m "Projektstruktur angelegt"
git push

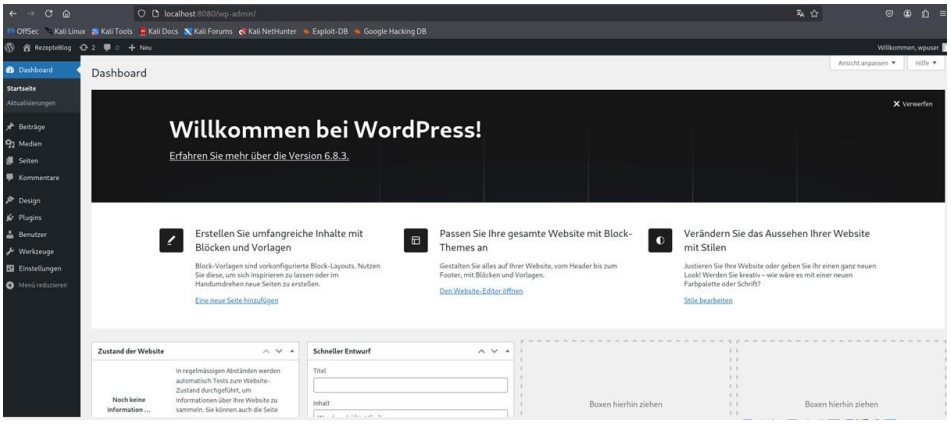


System bauen s automatisieren (Realisierung R-K)

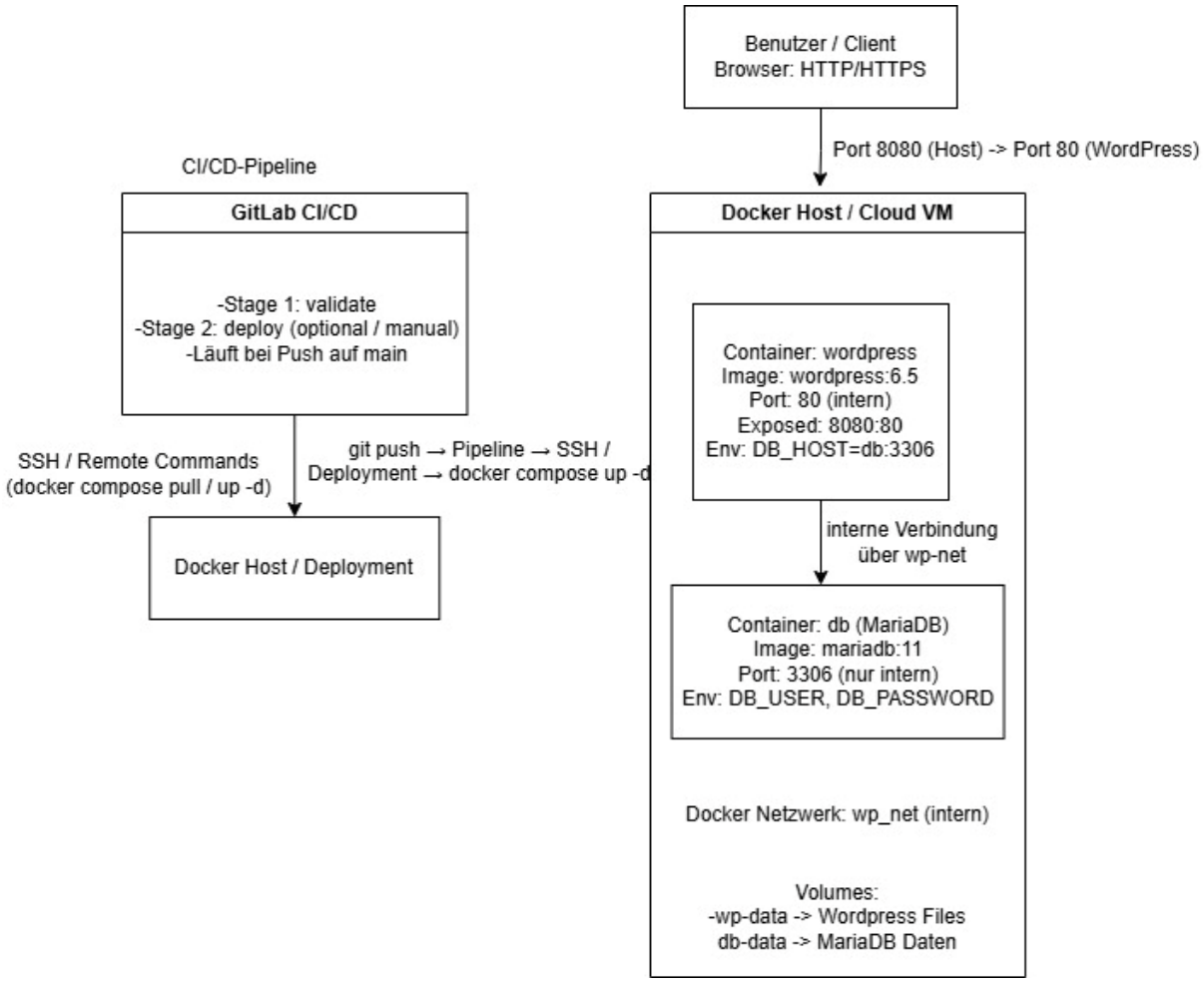
1. docker-compose.yml im Ordner Projekt/infra/ anlegen.
2. Starten:
   cd Projekt/infra
   docker compose up -d
   wir haben das file im dateien explorer angepasst und gespeichert
3. Weboberfläche via http://localhost:8080 prüfen







Erste Infrastruktur:



Infrastrukturübersicht

Die erste Infrastruktur basierte auf einer containerisierten Umgebung mit zwei Hauptkomponenten: einem WordPress-Webserver und einer MariaDB-Datenbank. Beide wurden als separate Docker-Container auf einem gemeinsamen Docker-Host betrieben und über Docker Compose verwaltet.

Die Container kommunizierten über ein internes Docker-Netzwerk. Der WordPress-Webserver war über Port 80 erreichbar, welcher am Host auf Port 8080 veröffentlicht wurde. Die Datenbank war gedacht ausschließlich intern erreichbar und nicht nach aussen exponieren.

Zur Sicherstellung der Datenpersistenz wurden Docker-Volumes eingesetzt. WordPress-Dateien sowie die Datenbankinhalte blieben dadurch auch nach einem Neustart der Container erhalten.

Erste CI/CD-Pipeline Grundgerüst bauen:

Wechsel des Projekts während der Umsetzung

Ursprünglich war geplant, die Aufgabenstellung mit einer neuen WordPress-Umgebung umzusetzen. Während der Projektphase haben wir jedoch entschieden, stattdessen ein bereits begonnenes eigenes Projekt weiterzuführen und dafür die CI/CD-Anforderungen zu integrieren.
Der Grund dafür war, dass unser bestehendes Projekt (GlobeNotes) bereits eine funktionierende Frontend- und Backend-Struktur hatte, wodurch wir die Pipeline-Themen praxisnäher und sinnvoller anwenden konnten.
Somit konnten wir die Lernziele des Moduls vollständig erfüllen, gleichzeitig aber ein realistischeres und für uns hilfreicheres Anwendungsszenario nutzen.

Entgültige Projektidee:

Vorgaben

Die Anwendung besteht aus zwei Hauptkomponenten: der Webapplikation und einer Persistenten H2 Datenbank. Beide Komponenten werden mittels Docker Containern betrieben und über eine gemeinsame docker-compose-Konfiguration verwaltet.
Das Deployment wird zusätzlich durch eine CI/CD-Pipeline (GitLab CI) unterstützt, welche die Konfiguration validiert und ein automatisiertes Deployment ermöglicht.

Was soll unsere App sein?

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

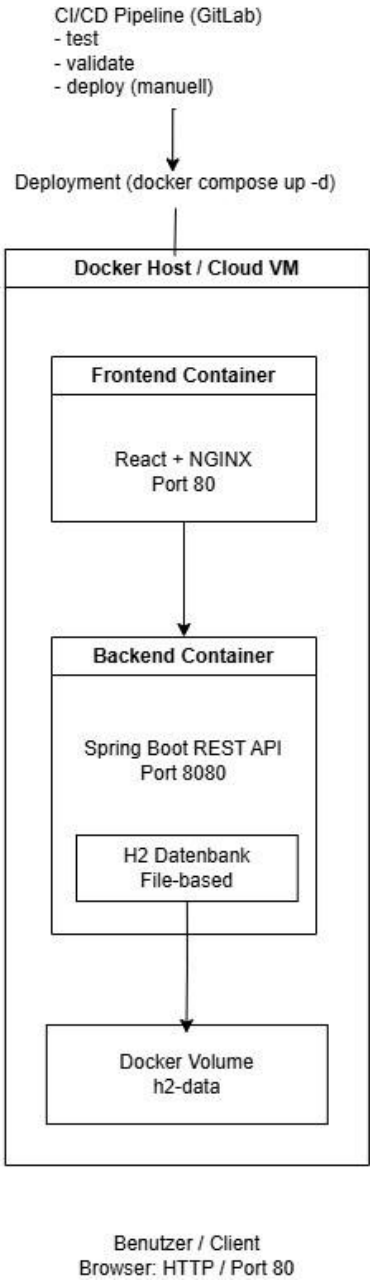
- Reiseziele erfassen, anzeigen und löschen
- Kategorien C Reisearten filtern
- Bilder speichern
- Daten bestehen auch nach Neustart dank persistenter Datenhaltung

Technisch wurde das System so umgesetzt,
dass es vollständig in Containern läuft und automatisiert deployt werden kann.

Der Fokus liegt bewusst auf:

- Frontend–Backend-Kommunikation
- Deployment über Docker Compose
- CI/CD-Pipeline in GitLab
- Persistente Datenbank (H2)

Entgültige Infrastruktur:



Das Infrastrukturdiagramm zeigt den vollständigen Aufbau der Applikation *GlobeNotes* inklusive CI/CD-Pipeline, Container-Architektur und Datenpersistenz.

Die Bereitstellung erfolgt über eine GitLab-CI/CD-Pipeline, welche bei einem Push auf den Branch *main* automatisch ausgelöst wird.
Während der Pipeline werden Frontend und Backend gebaut, die Docker-Compose-Konfiguration validiert und das Deployment über docker compose up -d ausgeführt.

Die Applikation läuft auf einem Docker Host.
Das Frontend wird in einem eigenen Container betrieben und über einen NGINX-Webserver auf Port 80 ausgeliefert.
Der Benutzer greift über einen Webbrowser darauf zu.

Das Backend läuft in einem separaten Container und stellt eine Spring-Boot-basierte REST-API auf Port 8080 bereit.
Innerhalb des Backend-Containers wird eine H2 File-basierte Datenbank verwendet.
Die Datenbank speichert ihre Daten in einem Docker Volume, wodurch eine persistente Speicherung auch nach Neustart der Container sichergestellt ist.
Die gesamte Kommunikation zwischen den Containern erfolgt über ein internes Docker-Netzwerk.

Kommunikation

Der Benutzer greift über einen Webbrowser auf das Frontend der Applikation zu.
Das Frontend wird über einen NGINX-Webserver im Frontend-Container auf Port 80 ausgeliefert.
API-Anfragen werden vom Frontend per HTTP an das Backend (Spring Boot) auf Port 8080 gesendet.
Das Backend verarbeitet die Anfragen und greift für persistente Daten auf eine file-basierte H2-Datenbank zu, welche innerhalb eines Docker-Volumes gespeichert ist.
Die Kommunikation zwischen Frontend und Backend erfolgt über das interne Docker-Netzwerk.

Entgültige 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
h2-data (Volume)	Persistente Speicherung der H2-Datenbank
uploads (Volume)	Speicherung von hochgeladenen Bildern

Ports

Komponente	Port
Frontend	80
Backend	8080

docker-compose.yml (Auszug)

```
services:
 globenotes-backend:
  build: ../GlobeNotes/backend
  ports:
   - "8080:8080"
  volumes:
   - h2-data:/app/data
   - uploads:/uploads

 globenotes-frontend:
  build:  ../GlobeNotes
  ports:
```

```
    - "80:80"
  depends_on:
    - globenotes-backend

volumes:
 h2-data:
 uploads:
```

Durch diese Konfiguration werden Frontend und Backend gemeinsam gestartet.
Die Volumes stellen sicher, dass Daten auch nach einem Neustart der Container erhalten bleiben.

Backend Konfiguration (Spring Boot)

Das Backend basiert auf **Spring Boot** und wird als ausführbares JAR innerhalb eines Docker-Containers betrieben.
Wichtige Konfigurationsparameter (Auszug)

```
server.port=8080

spring.datasource.url=jdbc:h2:file:./data/reiseziele
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.settings.web-allow-others=true
```

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.

CI/CD Konfiguration (GitLab)

Für die Automatisierung des Build- und Deployment-Prozesses wird **GitLab CI/CD** verwendet.

Bei einem Push oder Merge Request auf den Branch *main* wird automatisch eine GitLab CI/CD-Pipeline gestartet.
In der ersten Stage werden Frontend und Backend gebaut, um sicherzustellen, dass der Code fehlerfrei ist.
Anschliessend wird die Docker-Compose-Konfiguration validiert, um syntaktische Fehler auszuschliessen.

Optional kann anschliessend ein manueller Deploy-Job ausgeführt werden, welcher die Applikation mittels docker compose up -d startet.
Dadurch wird sichergestellt, dass nur geprüfter Code deployed wird.

Testplan:

Der Testplan konzentriert sich auf die wichtigsten Kernfunktionen der Applikation: Deployment, Erreichbarkeit der Services, API-Funktionalität, Datenpersistenz sowie die CI/CD-Pipeline.

Nr.	Testfall	Schritte	Erwartetes Ergebnis	Resultat
1	Deployment via Docker Compose	1. Terminal öffnen 2. In Projekt/infra/ navigieren 3. docker compose up –build ausführen	Frontend- und Backend-Container starten ohne Fehler, Volumes werden erzeugt	Gut
2	Frontend erreichbar	1. Browser öffnen 2. http://localhost aufrufen	GlobeNotes-UI wird angezeigt	Gut
3	Backend erreichbar	1. Browser öffnen 2. http://localhost:8080 aufrufen	Spring Boot liefert Whitelabel-Fehlerseite → zeigt Backend läuft	Gut
4	API-Funktion erreichbar	1. Browser öffnen 2. http://localhost:8080/reiseziele/ aufrufen	JSON-Antwort oder leeres Array wird angezeigt	Gut
5	CRUD-Funktion – Löschen	1. Frontend öffnen 2. Ein Reiseziel löschen 3. Seite aktualisieren	Eintrag ist nicht mehr sichtbar, kein Fehler angezeigt	Gut
6	Datenpersistenz (H2 Volume)	1. Ein oder mehrere Einträge erstellen 2. docker compose down + docker compose up -d ausführen	Einträge sind nach Neustart weiterhin sichtbar (Persistenz funktioniert)	Gut
7	H2-Konsole erreichbar	1. Browser öffnen 2. http://localhost:8080/h2-console aufrufen 3. Datenbankverbindung testen	Tabellen sichtbar, Verbindung erfolgreich	Gut
8	Pipeline-Build C Tests	1. Merge Request erstellen 2. CI ausführen lassen	Frontend-Build und Backend-Build laufen erfolgreich	Gut

Installationsanleitung:

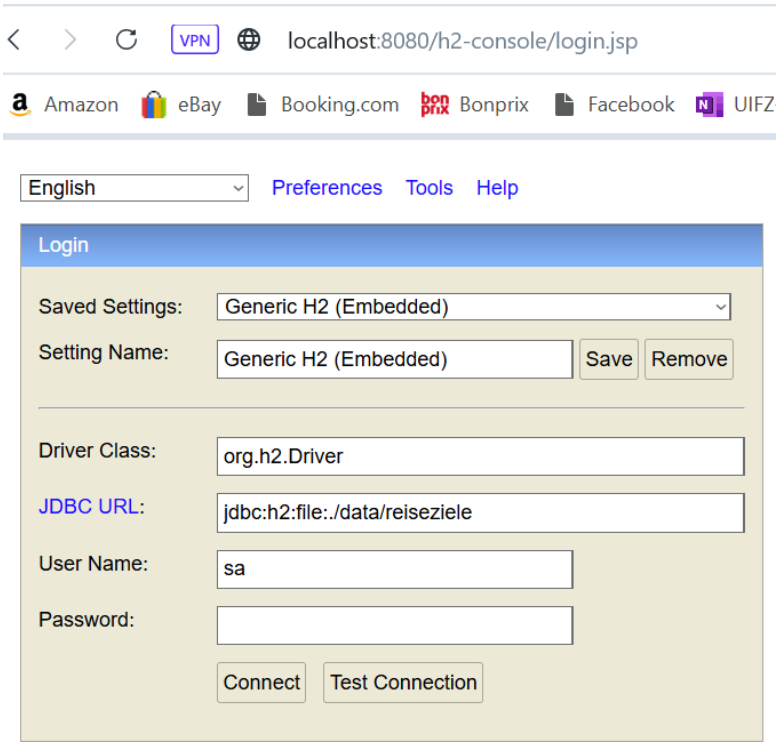
1. Voraussetzungen

- Docker Desktop installiert
- GitLab Repo geklont

2. Deployment (Docker)

Docker desktop starten
cd infra
docker compose up --build

Danach erreichbar via:
http://localhost → Frontend
http://localhost:8080 → Backend
http://localhost:8080/h2-console → JDBC URL : jdbc:h2:file:./data/reiseziele



Hilfestellung:

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