FH University of Applied Sciences TECHNIKUM WIEN

Dokumentation

SWEN

SS 2024

Name: Emre Koc, Nathaniel Ace Panganiban

Link zu Git-Repo:

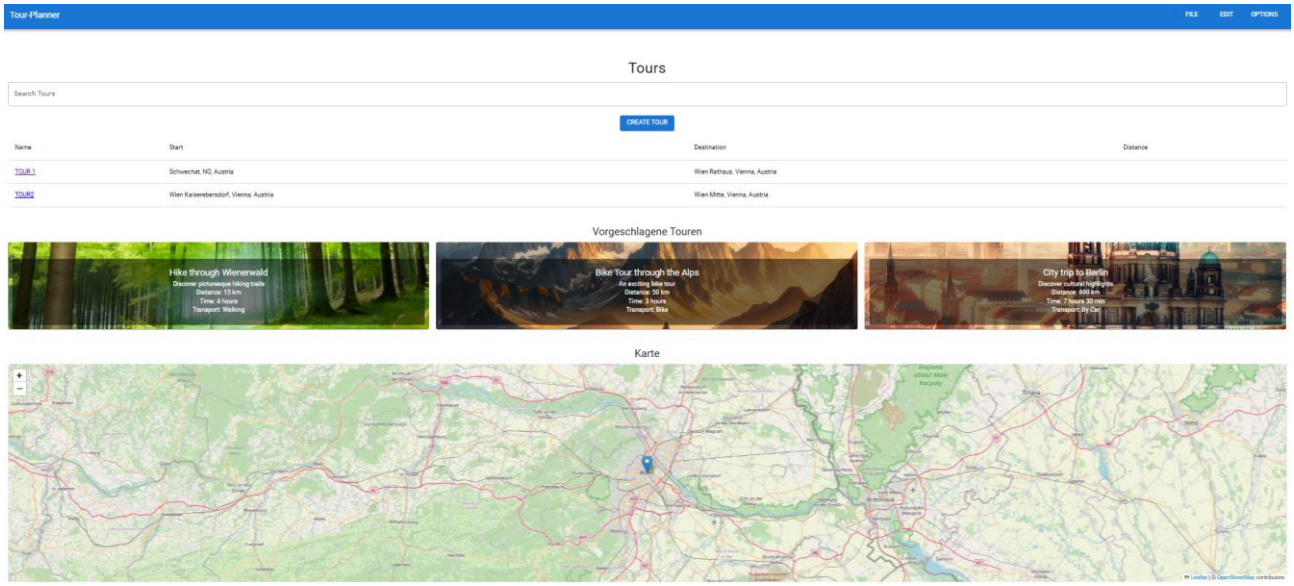
https://github.com/Nathaniel-Ace/Tour-Planner/tree/main

Protokoll der technischen Schritte und Entscheidungen

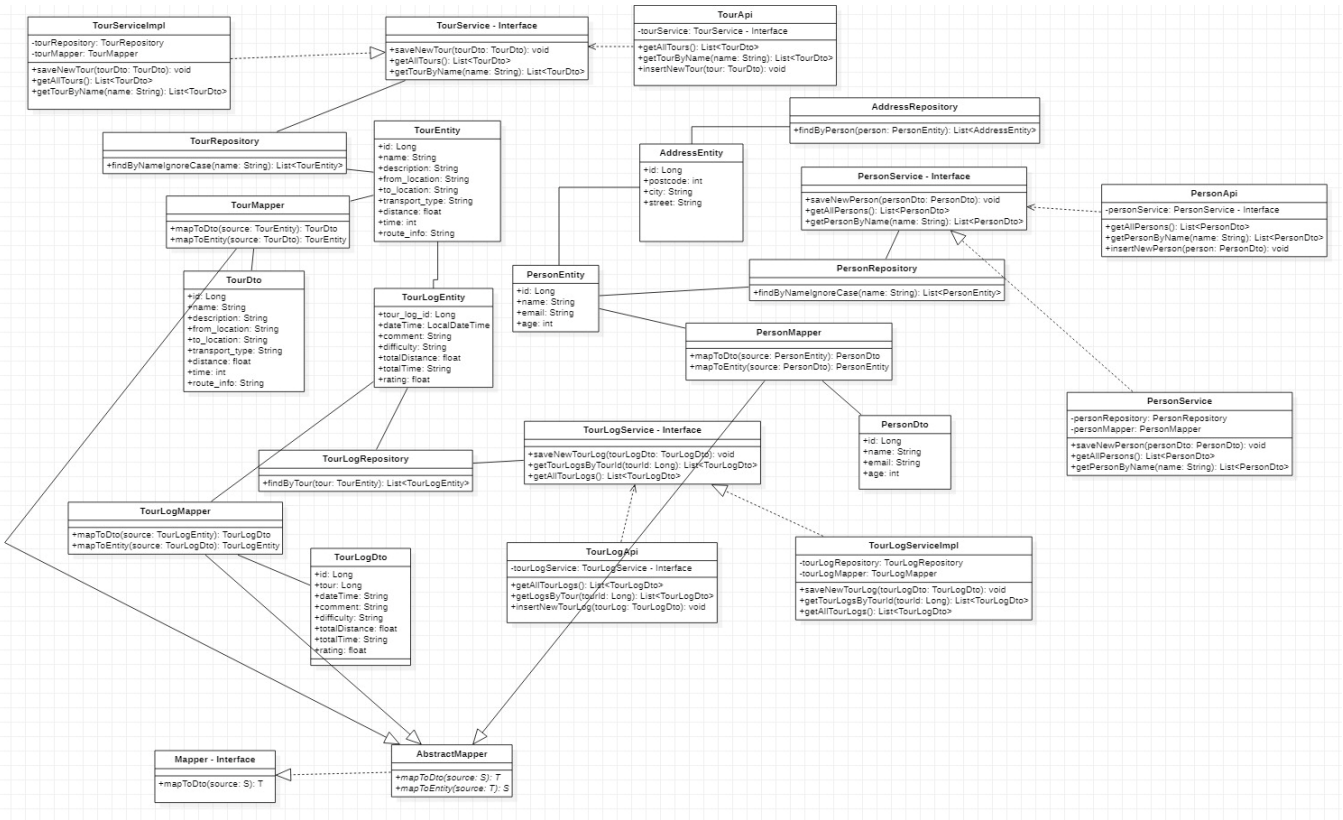
Designentscheidungen und ausgewählte Lösungen

In unserem Projekt "Tour Planner" haben wir mehrere wichtige technische Entscheidungen getroffen und spezifische Designansätze gewählt, um eine robuste und wartbare Anwendung zu erstellen. Hier sind die wesentlichen Punkte:

1. **Architekturentwurf**:
    - Wir entschieden uns für eine monolithische Architektur mit einer klaren Trennung von Backend und Frontend.
    - Das Backend wurde nach dem Schichtenmuster strukturiert, mit klar getrennten Schichten für API, Service und Persistence.
    - Das Frontend wurde mit einer komponentenbasierten Architektur unter Verwendung von React umgesetzt.
2. **Entwicklungsumgebung**:
    - Wir verwendeten IntelliJ IDEA als unsere Haupt-IDE zur Entwicklung der Anwendung.
    - Für die Frontend-Entwicklung verwendeten wir React, um eine dynamische und reaktionsfähige Benutzeroberfläche zu erstellen.



UML


TourServiceImpl
-tourRepository: TourRepository
-tourMapper: TourMapper
+saveNewTour(tourDto: TourDto): void
+getAllTours(): List<TourDto>
+getTourByName(name: String): List<TourDto>
TourService - Interface
+saveNewTour(tourDto: TourDto): void
+getAllTours(): List<TourDto>
+getTourByName(name: String): List<TourDto>
TourApi
-tourService: TourService - Interface
+getAllTours(): List<TourDto>
+getTourByName(name: String): List<TourDto>
+insertNewTour(tour: TourDto): void
AddressRepository
+findByPerson(person: PersonEntity): List<AddressEntity>
TourRepository
+findByNameIgnoreCase(name: String): List<TourEntity>
TourEntity
+id: Long
+name: String
+description: String
+from_location: String
+to_location: String
+transport_type: String
+distance: float
+time: int
+route_info: String
AddressEntity
+id: Long
+postcode: int
+city: String
+street: String
PersonService - Interface
+saveNewPerson(personDto: PersonDto): void
+getAllPersons(): List<PersonDto>
+getPersonByName(name: String): List<PersonDto>
PersonApi
-personService: PersonService - Interface
+getAllPersons(): List<PersonDto>
+getPersonByName(name: String): List<PersonDto>
+insertNewPerson(person: PersonDto): void
TourMapper
+mapToDto(source: TourEntity): TourDto
+mapToEntity(source: TourDto): TourEntity
TourDto
+id: Long
+name: String
+description: String
+from_location: String
+to_location: String
+transport_type: String
+distance: float
+time: int
+route_info: String
TourLogEntity
+tour_log_id: Long
+dateTime: LocalDateTime
+comment: String
+difficulty: String
+totalDistance: float
+totalTime: String
+rating: float
PersonEntity
+id: Long
+name: String
+email: String
+age: int
PersonRepository
+findByNameIgnoreCase(name: String): List<PersonEntity>
PersonMapper
+mapToDto(source: PersonEntity): PersonDto
+mapToEntity(source: PersonDto): PersonEntity
PersonService
-personRepository: PersonRepository
-personMapper: PersonMapper
+saveNewPerson(personDto: PersonDto): void
+getAllPersons(): List<PersonDto>
+getPersonByName(name: String): List<PersonDto>
PersonDto
+id: Long
+name: String
+email: String
+age: int
TourLogService - Interface
+saveNewTourLog(tourLogDto: TourLogDto): void
+getTourLogsByTourId(tourId: Long): List<TourLogDto>
+getAllTourLogs(): List<TourLogDto>
TourLogRepository
+findByTour(tour: TourEntity): List<TourLogEntity>
TourLogMapper
+mapToDto(source: TourLogEntity): TourLogDto
+mapToEntity(source: TourLogDto): TourLogEntity
TourLogDto
+id: Long
+tour: Long
+dateTime: String
+comment: String
+difficulty: String
+totalDistance: float
+totalTime: String
+rating: float
TourLogApi
-tourLogService: TourLogService - Interface
+getAllTourLogs(): List<TourLogDto>
+getLogsByTour(tourId: Long): List<TourLogDto>
+insertNewTourLog(tourLog: TourLogDto): void
TourLogServiceImpl
-tourLogRepository: TourLogRepository
-tourLogMapper: TourLogMapper
+saveNewTourLog(tourLogDto: TourLogDto): void
+getTourLogsByTourId(tourId: Long): List<TourLogDto>
+getAllTourLogs(): List<TourLogDto>
Mapper - Interface
+mapToDto(source: S): T
AbstractMapper
+mapToDto(source: S): T
+mapToEntity(source: T): S

Tour Planner Projektarchitektur

Unsere "Tour Planner" Anwendung ist mit einer monolithischen Architektur entworfen, die in eine klare und wartbare Struktur organisiert ist. Diese Architektur besteht aus einem Backend, das nach dem Schichtenmuster aufgebaut ist, und einem Frontend, das einen komponentenbasierten Ansatz verwendet.

Backend-Architektur

Das Backend unserer "Tour Planner" Anwendung ist in mehrere eindeutige Schichten strukturiert, die jeweils für spezifische Funktionalitäten verantwortlich sind, um eine klare Trennung der Zuständigkeiten zu gewährleisten.

1. **API Layer**
    - **Komponenten**: MapApi, TourApi, TourLogApi
    - **Rolle**: Stellt RESTful-Endpunkte bereit, um HTTP-Anfragen in Bezug auf Karten, Touren und Tour-Logs zu bearbeiten.
2. **Persistence Layer**
    - **Entities**: TourEntity, TourLogEntity
        - **Rolle**: Repräsentiert die Datenmodelle für Touren und Tour-Logs.
    - **Repositories**: TourLogRepository, TourRepository
        - **Rolle**: Verwalten CRUD-Operationen und Datenbankinteraktionen für Tour-Logs und Touren.
3. **Service Layer**
    - **DTOs**: TourDto, TourLogDto
        - **Rolle**: Erleichtert den Datentransfer zwischen den Schichten.
    - **Implementierungen**: MapServiceImpl, TourLogServiceImpl, TourServiceImpl
        - **Rolle**: Enthält die Geschäftslogik für Kartendienste, Tour-Logs und Touren.
    - **Service-Schnittstellen**: MapService, TourLogService, TourService
        - **Rolle**: Definiert die Verträge für die Dienste.
4. **Main Application**
    - **Komponente**: TourPlannerApplication
    - **Rolle**: Einstiegspunkt der Backend-Anwendung, verantwortlich für die Initialisierung des Anwendungskontexts und der Komponenten.

Frontend-Architektur

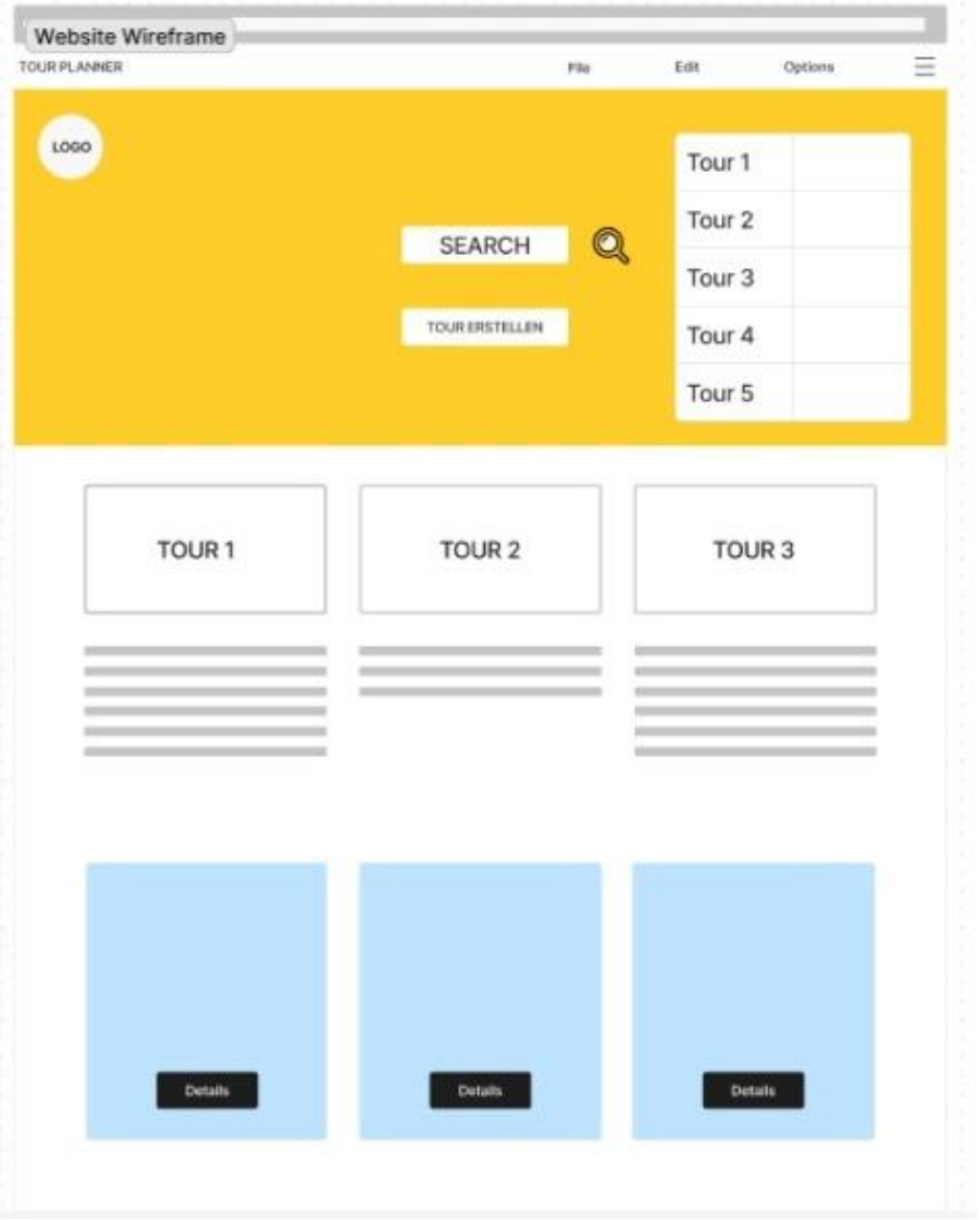
Das Frontend unserer "Tour Planner" Anwendung ist mit einer komponentenbasierten Architektur organisiert, die typisch für moderne Single-Page-Anwendungen (SPAs) ist, die mit React gebaut werden.

1. **Public Directory**
    - **Komponenten**: index.html, favicon.ico, logo192.png, logo512.png, manifest.json, robots.txt
    - **Rolle**: Enthält statische Ressourcen und die Haupt-HTML-Datei, die als Einstiegspunkt für die React-Anwendung dient.
2. **Source Directory**
    - **Komponenten**:
        - **Hauptkomponenten**: App.js, App.test.js
        - **Stile**: App.css, index.css
        - **Einstiegspunkt**: index.js
        - **Hilfsprogramme**: logo.svg, reportWebVitals.js, setupTests.js
    - **Rolle**: Enthält die React-Komponenten, die die Benutzeroberfläche definieren und Benutzerinteraktionen mit dem Backend verwalten.

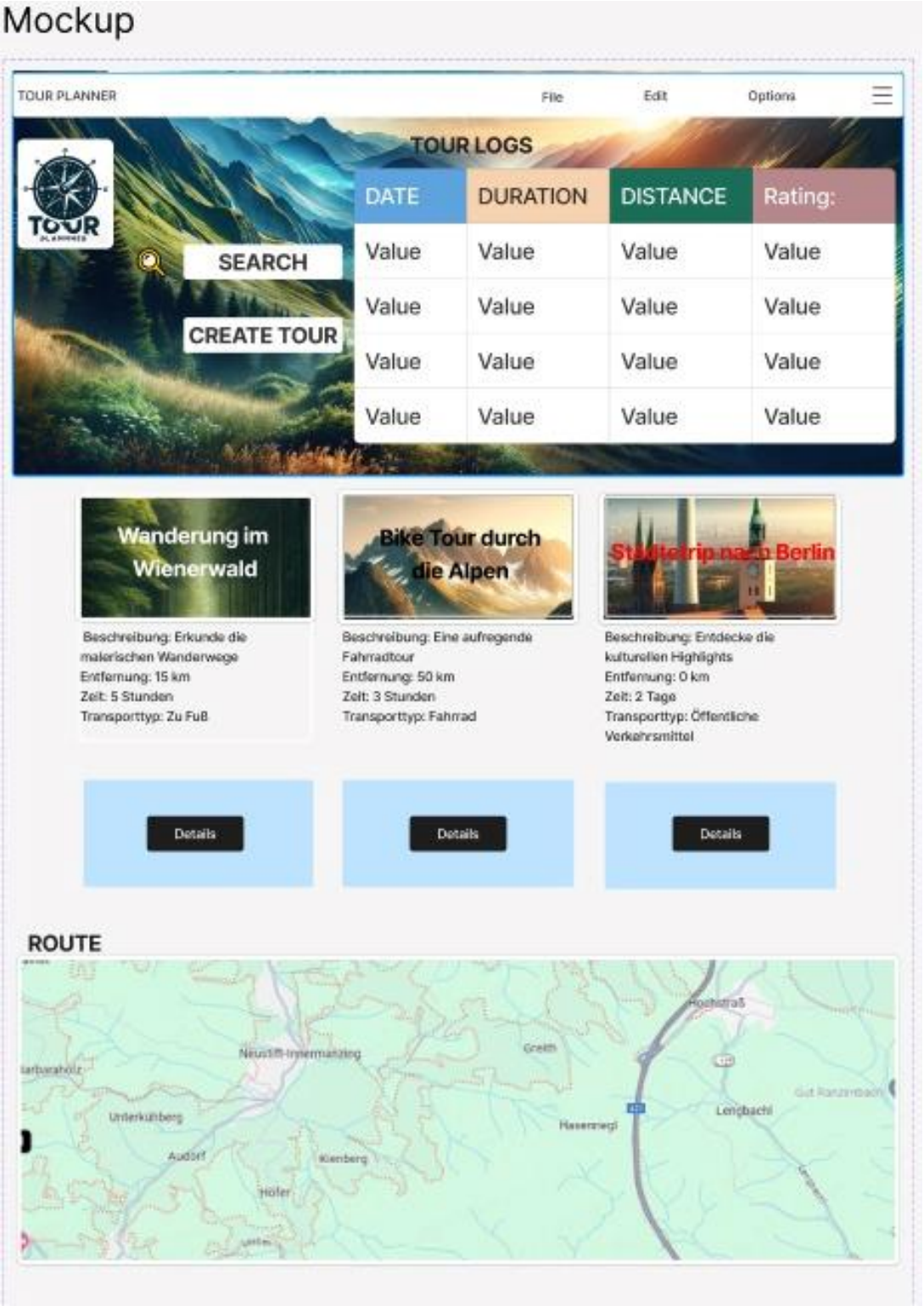
Zusammenfassung

Bei unserem Projekt "Tour Planner" verwenden wir eine monolithische Architektur mit einem geschichteten Backend und einem komponentenbasierten Frontend. Das Backend ist in API-, Persistenz- und Service-Schichten organisiert, um eine klare Trennung der Zuständigkeiten und eine wartbare Codebasis zu gewährleisten. Das Frontend nutzt eine komponentenbasierte Architektur, um eine modulare und wiederverwendbare Benutzeroberfläche zu erstellen. Dieser architektonische Ansatz gewährleistet Robustheit, Skalierbarkeit und Wartungsfreundlichkeit der Anwendung.

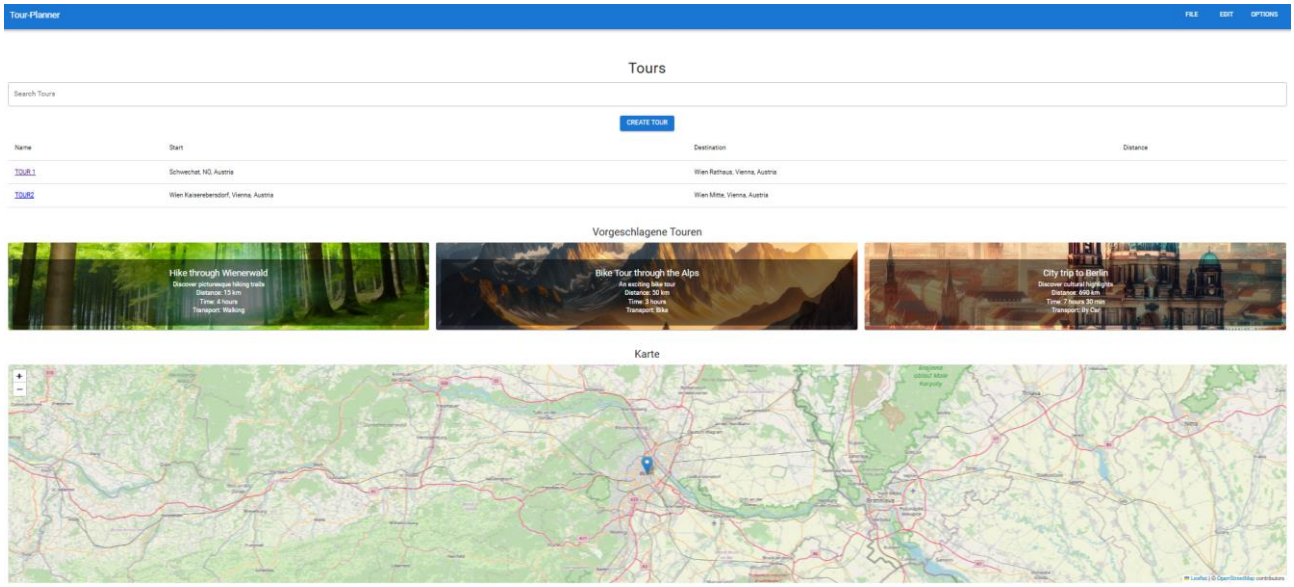
Wireframe


Website Wireframe
TOUR PLANNER
File
Edit
Options
LOGO
SEARCH
TOUR ERSTELLEN
Tour 1
Tour 2
Tour 3
Tour 4
Tour 5
TOUR 1
TOUR 2
TOUR 3
Details
Details
Details

Mockup


TOUR PLANNER
File
Edit
Options
TOUR LOGS
DATE
DURATION
DISTANCE
Rating:
Value
SEARCH
CREATE TOUR
Wanderung im Wienerwald
Bike Tour durch die Alpen
Beschreibung: Erkunde die malerischen Wanderwege
Entfernung: 15 km
Zeit: 5 Stunden
Transporttyp: Zu Fuß
Beschreibung: Eine aufregende Fahrradtour
Entfernung: 50 km
Zeit: 3 Stunden
Transporttyp: Fahrrad
Beschreibung: Entdecke die kulturellen Highlights
Entfernung: 0 km
Zeit: 2 Tage
Transporttyp: Öffentliche Verkehrsmittel
Details
ROUTE

Ergebnis



Technische Herausforderungen und Problemlösungen

Während der Entwicklung stießen wir auf mehrere Herausforderungen, die spezifische Lösungen erforderten:

Maven-Probleme:

- **Problem**: Nach einem Update der Projektdateien von GitHub wurden die Java-Klassen in IntelliJ nicht mehr korrekt angezeigt. Dies trat auf, nachdem mein Kollege Änderungen gepusht hatte und ich die Daten aktualisiert hatte.
- **Vermutete Ursache**: Wir vermuteten, dass das Problem auf unterschiedliche Maven-Versionen zurückzuführen war, die wir auf unseren Entwicklungsumgebungen installiert hatten.

Lösung:

1. **Anpassung der Ordnerstruktur**: Wir überprüften und korrigierten die Projektordnerstruktur in den IntelliJ-Einstellungen.
2. **Maven-Version**: Wir synchronisierten unsere Maven-Versionen, um Kompatibilitätsprobleme zu vermeiden.
3. **pom.xml-Datei**: Wir passten die pom.xml-Datei an, um sicherzustellen, dass alle Abhängigkeiten und Build-Konfigurationen korrekt definiert waren.

Erklärung der gewählten Unit-Tests

Die Unit-Tests in unserem Projekt decken verschiedene Aspekte der Anwendung ab. Hier ist eine kurze Erklärung, warum jeder dieser Tests gewählt wurde und warum der getestete Code kritisch ist:

1. **MapServiceTest**:
    - **Beschreibung**: Überprüft die Funktionalität des MapService.
    - **Warum kritisch**: Stellt sicher, dass Kartendaten korrekt verarbeitet und bereitgestellt werden, was für die Visualisierung von Touren essenziell ist.
2. **TourApiTest**:
    - **Beschreibung**: Testet die API-Endpunkte für Touren.
    - **Warum kritisch**: Gewährleistet, dass die API korrekt auf Anfragen reagiert und die richtigen Daten liefert, was für die Benutzerinteraktion wichtig ist.
3. **TourDtoTest**:
    - **Beschreibung**: Überprüft die Datenübertragungsobjekte (DTOs) für Touren.
    - **Warum kritisch**: Stellt sicher, dass die DTOs korrekt arbeiten und Daten zwischen Schichten sicher übertragen werden.
4. **TourEntityTest**:
    - **Beschreibung**: Testet die Entitäten für Touren.
    - **Warum kritisch**: Verifiziert, dass die Datenbankentitäten korrekt definiert und manipuliert werden, was für die Datenintegrität entscheidend ist.
5. **TourLogDtoTest**:
    - **Beschreibung**: Überprüft die DTOs für Tour-Logs.
    - **Warum kritisch**: Stellt sicher, dass die Datenübertragung für Tour-Logs korrekt funktioniert, um genaue Berichte und Statistiken zu liefern.
6. **TourLogEntityTest**:
    - **Beschreibung**: Testet die Entitäten für Tour-Logs.
    - **Warum kritisch**: Verifiziert die korrekte Definition und Manipulation der Tour-Log-Datenbankentitäten.
7. **TourLogServiceTest**:
    - **Beschreibung**: Überprüft die Geschäftslogik für Tour-Logs.
    - **Warum kritisch**: Stellt sicher, dass die Verarbeitung von Tour-Logs korrekt und konsistent erfolgt.
8. **TourLogTest**:
    - **Beschreibung**: Allgemeine Tests für Tour-Log-Funktionalitäten.

- **Warum kritisch**: Gewährleistet, dass die Tour-Log-Funktionalitäten umfassend und korrekt getestet werden.

9. **TourServiceTest**:
   - **Beschreibung**: Testet die Geschäftslogik für Touren.
   - **Warum kritisch**: Stellt sicher, dass die Tour-Verarbeitung korrekt und effizient erfolgt.
10. **TourTest**:
    - **Beschreibung**: Allgemeine Tests für Tour-Funktionalitäten.
    - **Warum kritisch**: Gewährleistet die umfassende und korrekte Funktionalität der Tourenverwaltung.