htlperg
innovativ | kreativ | praxisnah

Implementierung eines Ticketingsystems für das Donaudampfschiff Schönbrunn

DIPLOMARBEIT

Höhere Abteilung für Informatik

25/08/2025 – 26/03/2026

Projektmitglieder: Michael Lintner
Nicolas Müller
Simon Schatzl
Thomas Führer

Betreuer: OStR Dipl.-Ing. Dietmar Wokatsch-Ratzberger

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Bei der Erstellung der Arbeit habe wir die generativen KI-Tools zu folgendem Zweck verwendet: siehe Anhang A

Michael Lintner	
Nicolas Müller	
Simon Schatzl	

1 Danksagung

Wir, Michael Lintner, Simon Schatzl, Nicolas Müller und Thomas Führer, möchten im Rahmen unserer Arbeit zum Projekt „TicketShip“ unseren herzlichsten Dank aussprechen.

Unser besonderer Dank gilt Prof. Dipl.-Ing. Dietmar Wokatsch-Ratzberger. Durch seine geduldige Betreuung, seine fachliche Kompetenz, seine stetige Begleitung und seine wertvollen Inputs hat er wesentlich zum Gelingen dieses Projekts beigetragen. Seine konstruktive Kritik und kontinuierliche Unterstützung waren für uns während des gesamten Arbeitsprozesses sehr hilfreich.

Ebenso danken wir der Österreichischen Gesellschaft für Eisenbahngeschichte (ÖGEG), die es uns ermöglicht hat, dieses Projekt in einem realen Umfeld umzusetzen. Speziell erwähnen möchten wir MMAG. Dr. Thiemo Gaisbauer, unseren persönlichen Betreuer und Ansprechpartner. Seine Unterstützung, die offene Kommunikation und die zur Verfügung gestellten Ressourcen haben uns die Arbeit erheblich erleichtert und die Umsetzung des Projekts unterstützt

Nicht zuletzt danken wir unseren Familien und Freunden für ihre Geduld, Motivation und ihren Glauben an unsere Arbeit. Ihre Unterstützung hat uns durch die intensiven Phasen des Projekts getragen und war für den Erfolg von großer Bedeutung.

2 Kurzfassung

2.1 Aufgabenstellung

Diese Abschlussarbeit beschreibt die Entwicklung einer Webanwendung, die in Zusammenarbeit mit der ÖGEG, der Österreichischen Gesellschaft für Eisenbahngeschichte, erstellt wurde. Ziel des Projekts ist es, den Buchungsprozess für Fahrkarten für eine Schiffsfahrt zu vereinfachen und dem Schaffner die Entwertung der Fahrkarten zu erleichtern. Um das Problem endloser Papierarbeit und unvorhersehbarer menschlicher Fehler zu beheben, können Nutzer ihre Fahrkarten nun über einen Webshop erwerben und erhalten daraufhin einen eindeutigen QR-Code. Der Schaffner kann die Daten des Tickets dann über unsere selbst entwickelte mobile App entwerten.

2.2 Realisierung

Die Realisierung umfasst eine Webschnittstelle, die von der ÖGEG gehostet wird. Die verwendeten Kerntechnologien sind React und CSS. Nutzer müssen zunächst eine gewünschte Strecke aus den auf der Website verfügbaren Optionen auswählen. Anschließend müssen sie die Anzahl der Passagiere pro Ticket angeben. Zuletzt geben sie persönliche Daten wie Name, E-Mail-Adresse und Telefonnummer ein. Das BackOffice läuft auf Jakarta EE Servlets mit JSP-Views. Darüber verwalten Administratoren Linien, Haltestellen und Veranstaltungen. Die Validierungs-App ist mit Ionic und Angular gebaut und spricht per HTTP mit dem Backend.

2.3 Ergebnis

Der Webshop sendet daraufhin eine Nachricht an die angegebene E-Mail-Adresse. Die E-Mail enthält einen QR-Code, der vom Schiffsschaffner mithilfe unserer App gescannt werden kann. Nach dem Scannen wird ein gültiges Ticket mit den korrekten Informationen angezeigt, das den Zugang zum Schiff ermöglicht. Im BackOffice lassen sich Linien, Veranstaltungen und Tickets verwalten, Reservierungen bestätigen oder ablehnen. Die Validierungs-App unterscheidet beim Scan anhand eines Präfixes zwischen Linien- und Eventticket und zeigt das Ergebnis farbcodiert an.

3 Abstract

This thesis describes the development of a web application that has been created in cooperation with the ÖGEG, Austrian Society for Railway History. The aim of the project is to simplify the process of booking tickets for a ride on the ship and to make it easier for the conductor to validate the tickets. To fix the problem of endless paperwork and unpredictable human errors, users can now purchase their tickets through a web-shop and are then granted a unique QR-Code. The conductor is then able to validate the ticket's data through our self-developed mobile-app

The implementation includes a web interface that is hosted by ÖGEG. The core development technologies that were used are React and CSS. Users first have to choose a desired line from those available on the website. Then, they have to select the number of passengers per ticket. Finally, they enter personal information, such as name, email address and phone number.

The web-shop then sends a message to the provided e-mail address. The e-mail contains a QR-Code that can be scanned by the ship's conductor using our app. Once scanned, a valid ticket with the correct information is presented, allowing access to the ship

Inhaltsverzeichnis

1 Danksagung **II**

2 Kurzfassung **I**

2.1 Aufgabenstellung . . . I

2.2 Realisierung . . . I

2.3 Ergebnis . . . I

3 Abstract **II**

3.1 Einleitung . . . 1

3.1.1 Motivation . . . 1

3.1.2 Zielsetzung . . . 1

3.2 Projektumfeld . . . 2

3.2.1 Projektteam . . . 2

3.2.2 Betreuung . . . 2

3.2.3 Auftraggeber . . . 3

4 Grundlagen und Methoden **4**

4.1 Verwendete Technologien . . . 4

4.1.1 Java . . . 4

4.1.2 JavaServer Pages (JSP) . . . 5

4.1.3 Ionic . . . 6

4.1.4 React . . . 8

4.1.5 Apache Maven . . . 10

4.1.6 Wildfly . . . 11

4.1.7 Docker Desktop . . . 12

4.2 Verwendete Entwicklungssysteme . . . 14

4.2.1 Visual Studio Code . . . 14

4.2.2 WebStorm . . . 16

4.2.3 IntelliJ IDEA . . . 16

4.2.4 GitLab . . . 17

4.2.5 Penpot . . . 18

4.3 Verwendete Bibliotheken und Plugins . . . 19
4.3.1 Node Package Manager (npm) . . . 19
4.3.2 JAX-RS . . . 20
4.3.3 JSON-B . . . 21
4.3.4 Hibernate . . . 21
4.3.5 jose4j . . . 21
4.3.6 jBCrypt . . . 21
4.3.7 Jakarta Mail . . . 22
4.3.8 Capacitor Barcode Scanner . . . 22
4.3.9 JSON Web Tokens . . . 22

5 Planung und Realisierung 24
5.1 Projektorganisation . . . 24
5.2 Projektstrukturplan . . . 25
5.2.1 Erklärung . . . 25
5.3 Gantt-Plan . . . 25
5.3.1 Erklärung . . . 25

6 Implementierung 27
6.1 Technischer Überblick . . . 27
6.2 Backend . . . 28
6.2.1 Utilities . . . 28
6.2.2 API . . . 29
6.2.3 Datenbank . . . 32
6.3 Frontend . . . 36
6.3.1 Login- und Registrierungs-Page . . . 36
6.3.2 Ticketbuchung . . . 42
6.3.3 Ticketvalidierungs-App (Ionic) . . . 52
6.3.4 BackOffice . . . 57

7 Ergebnis 65
7.1 Backend . . . 65
7.2 BackOffice . . . 65
7.3 Online Shop . . . 65
7.4 Validierungs App . . . 65

8 Resümee 67

8.1 Backend 67

8.2 BackOffice 67

8.3 Online Shop 67

8.4 Validierungs App 67

9 Aufgabenverteilung 68

9.1 Michael Lintner 68

9.2 Nicolas Müller 69

9.3 Simon Schatzl 70

9.4 Thomas Führer VI

Literaturverzeichnis VII

Abbildungsverzeichnis VIII

Quellcodeverzeichnis IX

Anhang X

A KI Nutzung X

3.1 Einleitung

3.1.1 Motivation

TicketShip wurde von der ÖGEG initiiert. Die ÖGEG ist ein Verein, der sich mit der Pflege, Erhaltung und dem Betrieb historischer Eisenbahnfahrzeuge beschäftigt. 1995 konnten sie das historische Schaufelraddampfschiff 'Schönbrunn', welches auf der Donau bei Linz in Oberösterreich unterwegs war, vor der Verschrottung retten und restaurieren.

Der Ablauf, ein Ticket für eine Fahrt auf dem Schiff zu kaufen und validieren zu lassen war mühsam und erforderte von Hand geschriebene Excel-Listen und stapelweise Zettel um alle Daten des Kaufs zu verfassen. Deshalb haben wir ein System entwickelt, das all diese Probleme löst. Mit einem übersichtlichen Online-Shop und einer schnellen und einfachen Ticket-Entwertung wird der Ablauf sowohl für Kunde und Mitarbeiter vereinfacht.

3.1.2 Zielsetzung

TicketShip ist eine Kombination aus einer Webapplikation und einer Mobile-App, welche die Möglichkeit bietet, das Ticket in Form eines QR-Codes zu visualisieren und relevante Daten zum Ticket mithilfe einer Kamera in der App zu validieren. Durch die Speicherung der Ticketinformation in einer Datenbank ist es nicht nötig die Tickets per Hand zu verwalten. Das effiziente Entwerten durch einen einfachen Scan verhindert auch mögliche Verspätungen des Schiffs, die dadurch entstehen könnten.

3.2 Projektumfeld

3.2.1 Projektteam

Unser Team besteht aus 4 Schülern der HTL Perg – Abteilung für höhere Informatik.

Michael Lintner

Projektleiter

Simon Schatzl

Projektmitglied

Nicolas Müller

Projektmitglied

3.2.2 Betreuung

Ein wichtiger Aspekt unseres Projekts war die kontinuierliche Betreuung durch Prof. Dipl.-Ing. Dietmar Wokatsch-Ratzberger.

Wenn ein Projektstatusmeeting stattfand, bei dem wir unseren Fortschritt präsentierten, nahm Prof. Wokatsch Ratzberger daran teil und unterstützte uns bei der Vorgehensweise, insbesondere in Bereichen, in denen Fragen aufkamen. Zusätzlich wurden wir monatlich anhand von Protokollen bewertet, in denen zusammengefasst wurde, welches Projektmitglied welches Arbeitspaket umgesetzt hatte und ob der Zeitplan dabei eingehalten wurde.

3.2.3 Auftraggeber

Die Österreichische Gesellschaft für Eisenbahngeschichte (ÖGEG) ist ein österreichischer Verein, der sich mit der Pflege, Erhaltung und dem Betrieb historischer Eisenbahnfahrzeuge beschäftigt. Sie wurde in den frühen 1970er-Jahren von einer Gruppe Eisenbahn-Fans gegründet, die die letzten noch aktiven Dampflokomotiven betreuen wollten und die Geschichte der Eisenbahn für die Zukunft bewahren wollten.

Abbildung 1: https://www.oegeg.at/termine-2026/termine-schmalspur-steyrtalbahn/

Heute betreibt die ÖGEG mehrere Projekte wie das Eisenbahn- und Bergbaumuseum im Lokpark Ampflwang, historische Museumsbahnen, darunter die Steyrtalbahn, und organisiert Sonderfahrten mit alten Dampf- und Diesellokomotiven. Die Gesellschaft kümmert sich auch um den Erhalt von historischen technischen Fahrzeugen und bietet so Eisenbahn-Fans die Möglichkeit, historische Züge noch live zu erleben. [1, 2]

4 Grundlagen und Methoden

4.1 Verwendete Technologien

4.1.1 Java

Für die Umsetzung des Backends haben wir uns bewusst für Java entschieden. Diese Entscheidung basierte auf unseren bisherigen Erfahrungen aus dem Unterricht, in dem wir intensiv mit Java gearbeitet haben. Durch diese vertraute Umgebung konnten wir auf bestehendes Wissen zurückgreifen und mögliche Stolpersteine, die bei einer neuen Programmiersprache aufgetreten wären, vermeiden.

Dank dieser Vorkenntnisse konnten wir die Architektur des Backends effizient planen und umsetzen. Java ermöglichte uns eine saubere Trennung von Logik und Datenverwaltung, was insbesondere bei komplexeren Projekten von großer Bedeutung ist. Durch den Einsatz bewährter Frameworks wie Spring konnten wir zudem wiederkehrende Aufgaben automatisieren, die Entwicklungszeit erheblich reduzieren und die Wartbarkeit des Systems deutlich verbessern. Auch die Integration von Datenbanken und die Handhabung von REST-Schnittstellen gestalteten sich durch die umfangreichen Bibliotheken und Community-Ressourcen von Java sehr unkompliziert.

Außerdem bot Java die nötige Stabilität und Performance, die für unser Projekt entscheidend waren. Besonders in Hinblick auf parallele Prozesse, Datenkonsistenz und Ausfallsicherheit zeigte sich Java als zuverlässig und robust. Die große Entwickler-Community und die umfangreiche Dokumentation unterstützten uns zusätzlich dabei, effizient Lösungen für auftretende Probleme zu finden.

Insgesamt war die Wahl von Java optimal, da sie auf unseren vorhandenen Kompetenzen aufbaute und uns gleichzeitig eine solide Grundlage für die Weiterentwicklung des Systems bot. Sie erlaubte es uns, das Backend nicht nur funktional, sondern auch strukturell hochwertig zu gestalten, sodass es langfristig erweiterbar und wartbar bleibt.

4.1.2 JavaServer Pages (JSP)

JavaServer Pages (JSP) ist eine von Oracle entwickelte Technologie, die es ermöglicht, dynamische Webseiten mit Java zu erstellen. JSP erlaubt es Java-Code direkt in HTML-Seiten einzufügen. Wenn eine JSP-Seite aufgerufen wird, wird sie serverseitig in ein Servlet umgewandelt und kompiliert, das dann die HTML-Ausgabe an den Browser des Nutzers sendet. So können Webseiten dynamisch auf Daten aus Datenbanken oder anderen Quellen zugreifen und deren Inhalte zur Laufzeit generieren.

Das Ziel von JSP ist es, die Entwicklung serverseitiger Webanwendungen zu erleichtern, indem Darstellung und Logik in einer HTML-Struktur kombiniert werden. Entwickler können dabei auf die gesamte Java-Plattform zugreifen, einschließlich vorhandener Java-Bibliotheken und Frameworks. JSP-Seiten laufen in einem Servlet-Container und profitieren von der Stabilität und Plattformunabhängigkeit der Java-Laufzeitumgebung.

Ein zentrales Konzept von JSP ist die Verwendung von Expression Language (EL) und der JSP Standard Tag Library (JSTL). Diese ermöglichen es, Daten aus dem Backend in die HTML-Ausgabe einzufügen, ohne dass umfangreicher Java-Code direkt in die Seite geschrieben werden muss. Dadurch wird eine klare Trennung von Darstellung und Logik erreicht. Zudem können eigene Tag-Bibliotheken erstellt werden, um wiederkehrende Funktionen zu kapseln und wiederverwendbar zu machen.

Ein weiteres Merkmal von JSP ist die nahtlose Integration in bestehende Java-Infrastrukturen. Da JSP-Seiten in Servlets kompiliert werden, können sie direkt mit anderen Java-EE-Komponenten wie Servlets, JDBC-Verbindungen oder Enterprise JavaBeans interagieren. Dies macht JSP besonders geeignet für Anwendungen, die bereits auf einer Java-basierten Backend-Architektur basieren.

JSP im Projekt 'TicketShip'

Im Projekt 'TicketShip' wird JSP für die Entwicklung des Backoffice eingesetzt. Das Backoffice dient der Verwaltung von Veranstaltungen, Tickets und weiteren administrativen Aufgaben. Durch die Verwendung von JSP konnte die Benutzeroberfläche des Backoffice direkt an das bestehende JDBC-basierte Backend angebunden werden, ohne eine zusätzliche Schnittstelle wie eine REST-API implementieren zu müssen. Die serverseitige Generierung der Seiten ermöglicht einen direkten Zugriff auf die Datenbankschicht, was die Entwicklung des Prototypen beschleunigte.

Die Vorteile von JSP für ´TicketShip´ sind zusammengefasst:

- Direkte Backend-Anbindung: JSP ermöglicht den unmittelbaren Zugriff auf das bestehende JDBC-Backend, ohne weiteren Entwicklungsaufwand
- Bewährte Technologie: JSP ist stabil, gut dokumentiert und Teil des Java-Ökosystems, auf dem das gesamte Backend aufbaut
- Schnelle Prototypenentwicklung: Durch die serverseitige Seitengenerierung konnten Verwaltungsoberflächen schnell umgesetzt werden.

4.1.3 Ionic

Ionic ist ein Open-Source-Framework, das die Entwicklung plattformübergreifender mobiler Anwendungen und Progressive Web Apps (PWAs) erleichtert. Es basiert auf Webtechnologien wie HTML, CSS und Javascript oder Typescript. Gegründet wurde das Framework 2013 von Drifty Co. (heute Ionic) und es wurde als Open-Source-Projekt veröffentlicht. Ionic lässt sich problemlos mit modernen JavaScript-Frameworks wie Angular, React oder Vue.js integrieren, sodass Entwickler die Technologien nutzen können, die sie bevorzugen. Ionic nutzt den Capacitor, um den Zugriff auf Gerätefunktionen wie Kamera, GPS und Dateisystem zu ermöglichen, während die Anwendungslogik und Benutzeroberfläche mit Webtechnologien erstellt werden.

Das Ziel von Ionic ist es, die Entwicklung mobiler Anwendungen zu vereinfachen, indem ein einziger Code für mehrere Plattformen, verwendet werden kann. Dadurch muss man nur eine Anwendung mit einem Entwicklerteam erstellen. Die resultierende Anwendung läuft in einer sogenannten WebView, einem eingebetteten Browser innerhalb der nativen App-Hülle. Außerdem kann diese Codebasis auch als Progressive Web App im Browser genutzt werden.

Ein zentrales Merkmal von Ionic ist die große Bibliothek an vorgefertigten UI-Komponenten, die das Aussehen nativer Plattformen nachahmen. Dazu gehören die Navigationsleisten, Listen, Tabs, Karten und weitere Bestandteile von Ionic. Diese Komponenten passen ihr Design automatisch der jeweiligen Plattform an. Das bedeutet das IOS Nutzer alles in vertrauten Stil sehen, und Android Nutzer auch alles in ihrem gewohnten Material-Design-Stil sehen. So erhalten die Nutzer ein vertrautes Bedienerlebnis, Auch wenn die Anwendung nicht nativ entwickelt wurde.

Ionic wird in verschiedenen Bereichen eingesetzt. Dazu gehören:

- Plattformübergreifende mobile Apps: Unternehmen nutzen Ionic, um mit einer einzigen Codebasis Anwendungen für Android, iOS und das Web gleichzeitig bereitzustellen.
- Unternehmensanwendungen: Ionic wird häufig für interne Business-Apps eingesetzt, etwa für Außendienst-Tools, Inventarverwaltung oder Mitarbeiterportale.

- Progressive Web Apps: Mit Ionic lassen sich auch PWAs entwickeln, die direkt im Browser laufen und offline-fähig sind, ohne Installation über einen App Store.

Ionic ist eines der etabliertesten Frameworks für die hybride App-Entwicklung und wird von namenhaften unternehmen wie Nasa, T-Mobile und Electronic Arts. Aufgrund der kontinuierlichen Weiterentwiclung von Capacitor und der Unterstützung verschiedener Frontend-Frameworks bleibt Ionic eine wichtige Technologie für die plattformübergreifende Anwendungsentwicklung. Es gibt keine Anzeichen dafür, dass die Unterstützung des Frameworks eingestellt wird, was die langfristige Wartbarkeit des Projekts sicherstellt.

Ionic im Projekt 'TicketShip'

Im Rahmen des Projektes 'TicketShip' Wird Ionic zusammen mit Angular verwendet, um die Kontrolleur-App zu entwickeln. Diese App ermöglicht es, Tickets vor Ort durch das Scannen von QR-Codes zu überprüfen. Nach dem Scannen werden die Ticketdaten mit dem Backend abgeglichen, um den Gültigkeitsstatus anzuzeigen. Über Capacitor wurde der native Zugriff auf die Gerätekamera ermöglicht, der für das Scannen der QR-Codes erforderlich ist.

Die Vorteile von Ionic für "TicketShip"lassen sich wie folgt zusammenfassen:

- Plattformunabhängigkeit: Eine einheitliche Codebasis für die Kontrolleur-App auf Android und iOS hat den Entwicklungsaufwand verringert.
- Nativer Gerätezugriff: Mit Capacitor konnte die Gerätekamera für das Scannen von QR-Codes integriert werden.
- Keine App-Store-Abhängigkeit: Die App kann direkt im Browser genutzt werden, was die Bereitstellung für Kontrolleure vereinfacht.
- Kosteneffizienz: Ionic ist, ähnlich wie React, kostenlos als Open-Source-Software verfügbar.

4.1.4 React

React, auch bekannt als React.js, ist eine JavaScript-Bibliothek zur Entwicklung von Benutzeroberflächen für Webseiten und Webanwendungen. Sie wird vor allem im Frontend eingesetzt, also in dem Teil einer Anwendung, den Benutzer direkt im Browser sehen und bedienen. React wurde ursprünglich von dem Facebook-Entwickler Jordan Walke entwickelt und erstmals 2011 innerhalb von Facebook verwendet. Im Jahr 2013 wurde die Bibliothek als Open-Source-Projekt veröffentlicht und wird seitdem von Meta (Facebook) sowie einer großen Entwickler-Community weiterentwickelt.

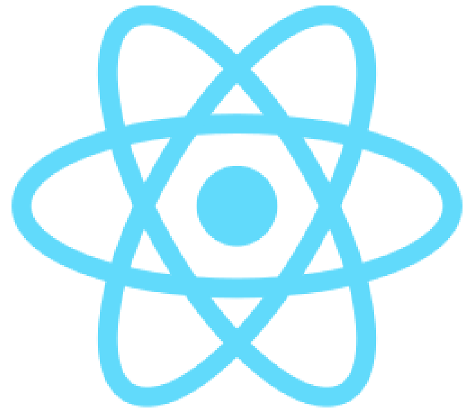

Abbildung 2: https://commons.wikimedia.org/wil icon.svg

React hat das Ziel, die Entwicklung moderner und interaktiver Webseiten einfacher und effizienter zu machen. Mit React können Entwickler Anwendungen erstellen, die sich schnell aktualisieren und dynamisch auf Änderungen von Daten reagieren, ohne dass die gesamte Seite neu geladen werden muss. Dadurch entstehen sogenannte Single-Page-Applications, bei denen Inhalte im Hintergrund aktualisiert werden und die Benutzeroberfläche flüssig bleibt.

Ein zentrales Konzept von React ist die Arbeit mit Komponenten. Eine Komponente ist ein wiederverwendbarer Baustein der Benutzeroberfläche, zum Beispiel ein Button, ein Formular oder eine Navigationsleiste. Diese Komponenten können kombiniert werden, um komplexe Benutzeroberflächen aufzubauen. Durch diese Struktur wird der Code übersichtlicher, leichter wartbar und einfacher wiederverwendbar.

Ein weiteres wichtiges Merkmal von React ist das sogenannte Virtual DOM. Dabei erstellt React eine virtuelle Darstellung der Webseite im Speicher und vergleicht Änderungen mit der echten Webseite im Browser. Anschließend werden nur die Teile aktualisiert, die sich wirklich verändert haben. Dieses Verfahren verbessert die Leistung von Webanwendungen und sorgt für schnelle Reaktionszeiten.

React wird heute in vielen Bereichen der Webentwicklung eingesetzt. Dazu gehören zum Beispiel:

- Single-Page-Applications (SPA): Viele moderne Web-Apps nutzen React, weil Inhalte schnell aktualisiert werden können, ohne die Seite neu zu laden.
- Interaktive Benutzeroberflächen: React wird häufig für komplexe Oberflächen verwendet, etwa bei Dashboards, Social-Media-Plattformen oder Online-Tools.

- Mobile Anwendungen: Mit React Native können auch mobile Apps für Android und iOS entwickelt werden, die auf ähnlichen Konzepten basieren.

React gehört heute zu den beliebtesten Technologien im Frontend-Bereich und wird von vielen bekannten Unternehmen verwendet, darunter Netflix, Airbnb oder Instagram. Durch die aktive Community und regelmäßige Updates bleibt React eine wichtige Technologie für moderne Webentwicklung.

React im Projekt ´TicketShip´

Das Projekt ´TicketShip´ nutzt React, um einen erweiterbaren Prototypen zu entwickeln. Die Flexibilität und Leistungsfähigkeit erleichtern das Entwickeln des Web-Shops.

Durch die Verwendung von React hatten wir die Möglichkeit, mit einem bereits bekannten Framework zu arbeiten, da wir dieses in vergangenen Projekten ebenfalls verwendet haben. Die Integration mit React-Router ermöglicht es uns zusätzlich, jegliche redirects auf andere Seiten zentral und dynamisch zu verwalten.

Der komponentenbasierte Aufbau von React hat dem Team sehr geholfen, die Website zu erweitern. Besonders hilfreich war dies nach Rückmeldungen von unserem Auftraggeber, wenn wir eventuell eine Komponente hinzufügen, anpassen oder löschen sollten.

Die Vorteile von React für ´TicketShip´ sind zusammengefasst:

- Erweiterbarkeit: Einfaches erweitern von Komponenten half dem ´TicketShip´-Team beim feinschliff
- Kosteneffizienz: Weil React kostenlos verfügbar ist.

[3] [4]

4.1.5 Apache Maven

Maven ist ein Tool, das dabei hilft, Java-Projekte zu verwalten. Es wurde von der Apache Software Foundation entwickelt. Es wurde 2004 zum ersten Mal veröffentlicht. Maven macht alles automatisch wenn eine Anwendung gebaut wird. In der Datei pom.xml findet man alle relevanten Informationen über das Projekt.

Maven ist praktisch, weil es Abhängigkeiten automatisch managt. Man muss die Bibliotheken nicht mehr manuell herunterladen und hinzufügen. Stattdessen gibt man in der pom.xml einfach die benötigte Abhängigkeit mit ihrem Gruppennamen, Artefaktnamen und der Version an. Maven lädt diese automatisch aus einem zentralen Repository herunter und fügt sie in das Projekt ein. Auch Bibliotheken die von einer eingebunden Bibliothek benötigt werden, werden automatisch aufgelöst.

Maven hat einen festen Ablauf für das Erstellen von Software. Dieser Ablauf hat mehrere Phasen. Die wichtigsten Phasen sind Compile, Test, Package und Install. Deshalb ist es einfach, sich in neue Projekte einzulernen. Maven kann um Funktionen erweitert werden, zum Beispiel um Anwendungen direkt auf einen Application Server zu bringen.

Maven hat viele Nutzungsbereiche:

- Maven ist das meist verwendete Werkzeug zum Erstellen von Java-Programmen und eignet sich für kleine und große Projekte. Das Einbinden von Bibliotheken ist mühsam. Mit dem Dependency-Management in der pom.xml ist das nicht mehr nötig. Maven kann man einfach in automatisierte Prozesse zur Erstellung und zum Einspielen von Software integrieren.
- Maven ist ein wichtiger Teil der Java-Entwicklung und wird von einer großen Community gepflegt. Es gibt zwar neuere Alternativen wie Gradle, aber Maven ist nach wie vor sehr wichtig für Java.

Maven im Projekt 'TicketShip'

Im Projekt 'TicketShip' benutzen wir Maven, um den gesamten Backend-Build zu verwalten. In der pom.xml stehen alle Abhängigkeiten des Projekts, zum Beispiel Jakarta EE, Hibernate, jose4j, jBCrypt und weitere Bibliotheken. Das Projekt wird als WAR-Datei verpackt und über das Wildfly-Maven-Plugin auf den Application Server gestellt. Wir testen mit dem Befehl mvn wildfly:dev. Dieser startet den Wildfly-Server und stellt die Anwendung automatisch bereit. So können wir Änderungen am Code schnell testen, ohne den Server manuell zu konfigurieren.

4.1.6 Wildfly

Wildfly ist ein Server für Java, der von Red Hat entwickelt wurde. Wildfly wurde 2014 von JBoss AS zu WildFly umbenannt. Der Server unterstützt Jakarta-EE-Spezifikation und bietet eine Laufzeitumgebung Java Webanwendung.

Wildfly funktioniert im Grunde wie ein Container, in den man seine fertig paketierte Anwendung als War Datei deployt. Der Server übernimmt dann die Bereitstellung von Datenbankverbindungen, Sicherheit, Transaktionen und HTTP-Schnittstellen. Dadurch müssen Entwickler diese Infrastruktur nicht selbst aufsetzen und konfigurieren.

Ein Merkmal von WildFly ist die modulare Architektur. Sie basiert auf Galleon Feature Packs. Man kann gezielt nur die Komponenten auswählen, die man braucht. Mit Layers kann man festlegen, welche Funktionen der Server bereitstellen soll, zum Beispiel JAX-RS, JSF oder eine MySQL-Datenbankanbindung. Dadurch bleibt der Server schlank und startet schneller.

Wildfly hat ebenfalls verschiedene Anwendungsbereiche:

- Anwendungen für Unternehmen: WildFly ist eine Plattform für geschäftliche Anwendungen, die auf Jakarta EE basieren.
- Webservices und APIs: WildFly ist dank JAX-RS hervorragend für REST-APIs geeignet.
- Maven-Plugins können für die Entwicklung und das Prototyping von Wildfly benutzt werden. Das geht direkt aus der Entwicklungsumgebung heraus. Und man kann es zum Testen verwenden.

Wildfly im Projekt 'TicketShip'

Wir benutzen Wildfly in Version 37.0.1 als Anwendungsserver für das Backend. Der Server wird nicht lokal installiert, sondern automatisch über das Wildfly-Maven-Plugin mit Galleon Feature Pack heruntergeladen und konfiguriert. Wir nutzen die Layers jaxrs-server, JSF, microprofile-platform und mysql-datasource. So enthält der Server genau die Komponenten, die unser Projekt benötigt. Wir starten den Server im Entwicklungsmodus, indem wir den Befehl 'mvn Wildfly:dev clean' eingeben. Dadurch werden Veränderungen im Code automatisch übernommen. Wildfly verbindet sich in unserem Projekt mit der MySQL-Datenbank und liefert die REST-API und die JSP-Seiten des Backoffice aus.

4.1.7 Docker Desktop

Docker Desktop ist eine Anwendung, die es ermöglicht, Software in sogenannten Containern auszuführen. Ein Container enthält alles, was eine Anwendung zum Laufen braucht: den Code, die Bibliotheken und die Einstellungen. Dadurch läuft die Software auf jedem Computer genau gleich, egal ob auf Windows, macOS oder Linux. Docker Desktop wurde von der Firma Docker entwickelt und vereinfacht die Verwaltung dieser Container über eine grafische Oberfläche.

Ein großer Vorteil von Docker ist die Trennung vom Betriebssystem. Man muss Datenbanken oder Webserver nicht mehr direkt auf dem eigenen Rechner installieren. Stattdessen startet man ein fertiges Image, das isoliert in einem Container läuft. Das hält das System sauber und verhindert Konflikte zwischen verschiedenen Versionen von Software.

Um mehrere Container gleichzeitig zu verwalten, nutzt man Docker Compose. In einer einfachen Textdatei (YAML-Format) wird definiert, welche Dienste gestartet werden sollen und wie sie miteinander verbunden sind. Mit nur einem Befehl lässt sich so eine komplette Infrastruktur inklusive Datenbanken und Netzwerkkonfiguration hochfahren.

Welche Vorteile bietet Docker Desktop

- **Konsistente Umgebung:** Entwickler arbeiten alle mit der exakt gleichen Konfiguration. Dadurch wird sichergestellt, dass die Anwendung in jeder Umgebung identisch ausgeführt wird, was Fehler durch abweichende lokale Systemkonfigurationen ausschließt.
- **Einfache Skalierbarkeit:** Container können schnell gestartet, gestoppt oder vervielfältigt werden, ohne das restliche System zu beeinflussen.
- **Großes Ökosystem:** Über den Docker Hub stehen tausende fertige Images für Datenbanken (wie MySQL), Webserver oder Tools sofort zur Verfügung.

Docker Desktop im Projekt 'TicketShip'

Im Projekt 'TicketShip' nutzen wir Docker Desktop vor allem, um die MySQL-Datenbank bereitzustellen. Anstatt MySQL manuell zu installieren und zu konfigurieren, nutzen wir die Datei `docker-compose.yml`. Diese Datei steuert den gesamten Prozess.

Hier ist unsere Konfiguration:

Listing 1: Docker Compose Konfiguration für TicketShip

```
1  version: "3.9"
2
3  services:
4  mysql:
5  image: mysql:latest
6  container_name: diploSQLServer
7  restart: unless-stopped
8
9  environment:
10   MYSQL_ROOT_PASSWORD: ************
11   MYSQL_DATABASE: ticketship
12
13 ports:
14   - "3306:3306"
15
16 # Persist data to a local folder
17 volumes:
18   - ./data:/var/lib/mysql
```

In dieser Datei definieren wir den Dienst `mysql`. Wir verwenden das aktuellste MySQL-Image und geben dem Container den festen Namen `diploSQLServer`. Über die `environment`-Variablen legen wir das Root-Passwort und den Namen der Datenbank fest, die beim Start automatisch erstellt wird.

Besonders wichtig sind die `ports` und `volumes`:

- Durch das Mapping `"3306:3306"` leiten wir den Datenbank-Port aus dem Container direkt an unseren Rechner weiter. So kann Wildfly auf die Datenbank zugreifen, als würde sie lokal laufen.
- Mit den `volumes` sorgen wir dafür, dass die Daten der Datenbank nicht verloren gehen, wenn der Container gestoppt wird. Sie werden im lokalen Ordner `./data` gespeichert.

4.2 Verwendete Entwicklungssysteme

4.2.1 Visual Studio Code

Visual Studio Code - meist VS Code genannt - wurde von Microsoft im Jahr 2015 veröffentlicht. Seitdem wurde es immer mehr zum Liebling bei Programmierern. Das liegt vor allem daran, dass der Editor nicht nur kostenlos ist, sondern auch plattformübergreifend auf Windows, macOS und Linux flüssig läuft.

Abbildung 3: https://de.wikipedia.org/wiki/Visual_Studio_Code

Was VS Code so beliebt macht, ist das übersichtliche Design und der breiten Auswahl an Funktionalitäten. Eine beliebte Funktion ist Syntax-Highlighting, wodurch der Code durch farbliche Markierung deutlich lesbarer wird. Hinzu kommt Intelli-Sense, wo automatisch Code vorgeschlagen wird aufgrund von den Eingaben des Benutzers. Zusätzlich unterstützt Visual Studio Code auch verschiedene Programmiersprachen von Python, C++, oder wie in unserem Projekt JavaScript. Somit besteht eine große Menge an Möglichkeiten um in VS Code zu programmieren.

Dank des eingebauten Terminals und der direkten Anbindung an Git kann man den gesamten Entwicklungsprozess, also vom Schreiben über das Testen bis hin zur Versionsverwaltung, an einem Ort erledigen. Man muss das Fenster also kaum noch verlassen, was im Arbeitsfluss enorm Zeit spart.

Visual Studio Code lässt sich zusätzlich mit verschiedenen Extensions anpassen. Diese Erweiterungen helfen beim Programmieren, da die Arbeit einfacher, schneller und oft auch schöner gemacht wird. Grundsätzlich unterscheidet man zwischen zwei Gruppen von Erweiterungen: Visuelle Extensions und funktionale Extensions.

Visuelle Extensions verändern, wie VSC aussieht oder wie der Code angezeigt wird. Sie machen die Arbeit angenehmer für die Augen und helfen dabei, den Überblick zu behalten. Ein Beispiel sind Themes, wie das One Dark Pro Theme, das den Editor dunkel färbt und die Schriftfarben gut lesbar macht. Andere visuelle Extensions sind Icon Packs wie vscode-icons, die Dateien mit unterschiedlichen Symbolen anzeigen, sodass man schnell erkennt, welche Art Datei es ist. Auch Bracket Pair Colorizer gehört dazu, welcher zusammengehörige Klammern in verschiedenen Farben einfärbt, was besonders bei komplizierten, verschachteltem Code hilft.

Funktionale Extensions erweitern die Möglichkeiten von VSC, sodass man mehr direkt im Editor machen kann, ohne externe Programme zu brauchen. Ein Beispiel ist Prettier, ein Code-Formatter, der automatisch den Code sauber formatiert. Live Server ist eine weitere beliebte funktionale Extension, welche einen kleinen lokalen Webserver startet und zeigt Änderungen am Code sofort im Browser an

Visual Studio Code wird besonders häufig verwendet für:

- Webentwicklung: Viele Entwickler nutzen Visual Studio Code zum Schreiben von HTML, CSS und JavaScript für Webseiten.
- Softwareentwicklung: Der Editor unterstützt viele Programmiersprachen und eignet sich für verschiedene Arten von Anwendungen.
- Cloud- und Backend-Projekte: Durch Erweiterungen kann Visual Studio Code auch für Server- und Cloud-Anwendungen eingesetzt werden.

Trotz vieler neuer Entwicklungswerkzeuge bleibt Visual Studio Code eine der wichtigsten Anwendungen für Programmierer. Durch regelmäßige Updates und eine große Entwickler-Community wird der Editor ständig weiterentwickelt und verbessert. Dadurch bleibt Visual Studio Code ein leistungsstarkes und flexibles Werkzeug für moderne Softwareentwicklung.

Visual Studio Code im Projekt ´TicketShip´

Visual Studio Code hat uns geholfen, in einer übersichtlichen IDE unseren Web-Shop zu implementieren. Da uns VS Code durchaus bekannt war, benötigte es keinen extra Lernaufwand um sich damit vertraut zu machen. Die direkte Verbindung mit GitHub erleichterte uns das Arbeiten auf mehreren Geräten und stellte somit kein Problem dar.

4.2.2 WebStorm

WebStorm ist eine Entwicklungsumgebung (IDE), die von JetBrains entwickelt wird und speziell für die Webentwicklung mit JavaScript und TypeScript gedacht ist. WebStorm wurde 2010 veröffentlicht und wird seitdem laufend verbessert. Im Gegensatz zu VS Code ist WebStorm keine leichtere Editor-Anwendung, sondern eine umfassende IDE, die viele Funktionen wie IntelliSense oder Git-Integration, ohne zusätzliche Plugins bietet.

Wie angesprochen ist ein wichtiges Merkmal von WebStorm die tiefe Integration von modernen Frameworks wie Angular und React. Dadurch kann die IDE automatisch die Projektstruktur erkennen und bietet sofort Code-Vervollständigung, Navigation und Refactoring-Tools, was den sich wiederholenden Aufwand minimiert. Auch die Syntaxanalyse ist sehr hilfreich wobei diese bereits in jeder IDE Standard ist. Besonders nützlich ist die native Unterstützung von TypeScript, was vor allem bei Angular-Projekten von Vorteil ist.

WebStorm wird häufig für folgende Zwecke eingesetzt:

- JavaScript- und TypeScript-Projekte: Webstorm bietet hierfür spezielle Werkzeuge und Analysen.
- Entwicklung mit Frameworks: Dank der nativen Unterstützung für Angular, React und Vue.js ist WebStorm ideal für Projekte, die auf Frameworks basieren.

WebStorm im Projekt 'TicketShip'

Für die Entwicklung der Kontrolleur-App mit Ionic und Angular haben wir WebStorm als unsere Entwicklungsumgebung gewählt. Da WebStorm Angular nativ unterstützt, konnten wir sofort von einer funktionierenden Code-Vervollständigung und Fehleranalyse für TypeScript profitieren, ohne zusätzliche Plugins installieren zu müssen. Die integrierten Refactoring-Tools haben uns besonders beim Umstrukturieren von Komponenten unterstützt. Auch die Git-Integration haben wir regelmäßig genutzt, um unsere Änderungen an der Kontrolleur-App zu verwalten.

4.2.3 IntelliJ IDEA

IntelliJ IDEA ist eine Entwicklungsumgebung (IDE), die von JetBrains entwickelt wird und hauptsächlich für die Programmierung mit Java gedacht ist. IntelliJ IDEA wurde 2001 veröffentlicht und zählt heute zu den bekanntesten IDEs im Bereich der Softwareentwicklung. Im Gegensatz zu einfachen Code-Editoren bietet IntelliJ IDEA eine umfangreiche integrierte

Umgebung, die viele Funktionen wie intelligente Code-Vervollständigung oder Git-Integration ohne zusätzliche Plugins bereitstellt.

Wie bereits erwähnt, ist ein zentrales Merkmal von IntelliJ IDEA die besonders starke Unterstützung für Java sowie verwandte Technologien wie Kotlin oder Spring. Die IDE erkennt automatisch die Projektstruktur und bietet sofort hilfreiche Funktionen wie Navigation, Refactoring-Tools und Fehleranalyse, wodurch sich wiederholende Aufgaben reduziert werden. Auch die statische Codeanalyse ist sehr ausgeprägt und hilft dabei, Fehler frühzeitig zu erkennen und zu beheben. Besonders nützlich ist außerdem die integrierte Unterstützung für Build-Tools wie Maven oder Gradle, was die Arbeit an größeren Projekten deutlich erleichtert.

IntelliJ IDEA wird häufig für folgende Zwecke eingesetzt: • Java- und Kotlin-Projekte: Die IDE bietet hierfür spezialisierte Werkzeuge und tiefgehende Analysen. • Backend-Entwicklung: Durch die Unterstützung von Frameworks wie Spring eignet sich IntelliJ IDEA ideal für serverseitige Anwendungen. • Allgemeine Softwareentwicklung: Dank Unterstützung vieler Programmiersprachen kann die IDE auch vielseitig eingesetzt werden.

4.2.4 GitLab

GitLab ist eine Plattform zur Versionsverwaltung, die auf dem Versionskontrollsystem Git, das von Linus Torvalds entwickelt wurde, basiert. Sie ist sowohl als Cloud-Lösung über gitlab.com als auch als selbst gehostete Instanz verfügbar wie bei unserem Projekt über den Schulserver. Neben der Repository-Verwaltung bietet GitLab auch Funktionen wie Merge Requests, Issue-Tracking und CI/CD-Pipelines.

GitLab hat den Vorteil, dass man damit Code zentral verwalten kann. Mehrere Entwickler können gleichzeitig an einem Projekt arbeiten. Änderungen werden in eigenen 'Branches' gemacht, welche später mit Merge Requests auf eine Codebasis zusammengeführt werden. So kann man sehen, wer welche Änderung wann vorgenommen hat. Die Oberfläche im Browser macht all das anschaulicher, aber die wichtigsten Funktionen sind grundsätzlich in der Konsole verfügbar. GitLab kann man auch auf eigenen Servern betreiben. Dadurch behalten Organisationen die volle Kontrolle über ihre Daten und können den Zugriff intern verwalten. Das ist vor allem in Schulen und Universitäten üblich. Dort werden keine sensiblen Daten auf externen Servern gespeichert.

Im Projekt 'TicketShip' nutzen wir eine Gitlab-Instanz zur Versionsverwaltung, die von der HTL Perg gehostet wird. Das gesamte Projekt wird in einem Repository organisiert, mit verschiedenen Unterordnern für die einzelnen Projektbestandteile, wie Buchungsplattform, Datenbank und

Kontrolleur-App. Wir nutzen GitLab hauptsächlich über das Terminal, außer bei Merge Requests. Wir benutzen GitLab, um Code zusammenzuführen und zu überprüfen. Alle Teammitglieder können auf dem Schulserver immer den aktuellen Stand des Projekts sehen.

4.2.5 Penpot

Penpot ist eine webbasierte Anwendung, die zur Entwicklung von Websites und mobilen Apps beiträgt. Sie ermöglicht es, Benutzeroberflächen intuitiv zu gestalten, ohne dass dafür Programmierkenntnisse erforderlich sind.

Während viele Design-Programme oft wie eine „Blackbox“ für Programmierer wirken, basiert Penpot auf offenen Standards wie **SVG und CSS**. Das bedeutet, dass das, was man auf der Oberfläche zeichnet, technisch viel näher an der späteren Realität im Browser liegt.

Der Einstieg ist denkbar simpel: Durch das Einfügen und Kombinieren von Grundformen wie Kreisen, Rechtecken oder Polygonen lassen sich in kürzester Zeit komplexe Oberflächen zusammenstellen. Dank der intuitiven **Drag-and-Drop-Steuerung** können Elemente völlig frei auf der Arbeitsfläche (dem „Canvas“) bewegt und angepasst werden. Ob es um die Rundung von Ecken, Farbverläufe oder Schatteneffekte geht – der Fantasie sind hier kaum Grenzen gesetzt.

Ein Punkt, der Penpot besonders sympathisch macht, ist sein **Open-Source-Charakter**. Im Gegensatz zu Platzhirschen wie Figma gehört Penpot der Community. Das heißt, man hat die volle Kontrolle über seine Daten und kann das Tool sogar auf eigenen Servern hosten, wenn man besonders viel Wert auf Datenschutz legt.

Warum man Penpot im Blick behalten sollte:

- **Flexibles Layout-System:** Mit Funktionen wie ´Flex Layout´ verhalten sich die Design-Elemente fast so wie echtes CSS. Wenn man einen Button größer macht, passt sich der Text darin automatisch an – genau wie auf einer echten Website.
- **Interaktive Prototypen:** Man kann einzelne Screens miteinander verknüpfen, um dem Kunden oder dem Team schon vorab zu zeigen, wie sich die App später anfühlen wird. Klickbare Buttons und Übergänge machen das Design lebendig.
- **Kostenlos und zugänglich:** Da es im Browser läuft, muss niemand schwere Software installieren. Ein Link reicht, und das ganze Team kann gleichzeitig am selben Entwurf arbeiten.

4.3 Verwendete Bibliotheken und Plugins

4.3.1 Node Package Manager (npm)

Für die Entwicklung des Frontends wurde der Node Package Manager (npm) als zentrales Werkzeug für die Paketverwaltung und die Steuerung des Build-Prozesses eingesetzt. npm ist der Standard-Paketmanager für die Laufzeitumgebung Node.js und ermöglicht es, externe Programmbibliotheken effizient in das Projekt zu integrieren, zu aktualisieren und zu verwalten.

1. Die Rolle der package.json Das gesamte Fundament der Frontend-Anwendung wird in der Datei `package.json` definiert. Sie fungiert als Inhaltsverzeichnis und Konfigurationsdatei des Projekts. Hier sind alle sogenannten „Dependencies“ (Abhängigkeiten) aufgelistet. Dazu gehören unter anderem:

- **React und React-DOM:** Die Kernbibliotheken für die Benutzeroberfläche.
- **React-Router-Dom:** Für die Navigation zwischen den verschiedenen Seiten (Login, Buchung, Zusammenfassung).
- **Web-Vitals:** Zur Messung der Performance der Anwendung.

Durch den Befehl `npm install` liest der Paketmanager diese Datei aus und lädt alle benötigten Bibliotheken automatisch in den Ordner `node_modules` herunter. Dies stellt sicher, dass die Entwicklungsumgebung auf jedem Rechner absolut identisch ist, was besonders bei der Arbeit im Team oder der Abgabe der Diplomarbeit entscheidend ist.

2. Verwaltung von Abhängigkeiten Ein großer Vorteil von npm ist die automatische Auflösung von Abhängigkeiten. Wenn eine Bibliothek eine andere Unter-Bibliothek benötigt, kümmert sich npm selbstständig darum, die korrekte Version zu finden. Dies verhindert Versionskonflikte, die bei einer manuellen Einbindung von Skripten häufig auftreten würden.

3. Skript-Steuerung und Workflow Neben der reinen Verwaltung von Paketen dient npm auch als Ausführungswerkzeug für den Lebenszyklus der Anwendung. In der Sektion `scripts` der `package.json` wurden Befehle hinterlegt, die den Workflow im Projekt massiv vereinfachen:

- **Entwicklungsmodus (`npm start`):** Dieser Befehl startet einen lokalen Webserver (meist unter `localhost:3000`). Der Code wird dabei durch `react-scripts` in Echtzeit überwacht. Sobald eine Datei gespeichert wird, kompiliert npm den Code neu und aktualisiert die Ansicht im Browser automatisch.

- **Build-Command (`npm run dev`):** Für die finale Abgabe oder das Deployment auf einem Server wird dieser Befehl genutzt. Er macht den gesamten JavaScript- und CSS-Code so klein und schnell wie möglich, indem er alle unnötigen Lücken, Leerzeichen und Kommentare entfernt, damit die Website später beim Nutzer besonders schnell lädt.. Dabei werden unnötige Kommentare und Leerzeichen entfernt, um die Ladezeiten für den Endnutzer so gering wie möglich zu halten.

Listing 2: Auszug aus der package.json Konfiguration

```
1  {
2    "name": "schoenbrunn-dampfschifffahrt",
3    "version": "2.0.0",
4    "private": true,
5    "type": "module",
6    "scripts": {
7      "dev": "vite",
8      "build": "vite build",
9      "preview": "vite preview"
10   },
11   "dependencies": {
12     "react": "^18.2.0",
13     "react-dom": "^18.2.0",
14     "react-router-dom": "^6.22.3"
15   },
16   "devDependencies": {
17     "@vitejs/plugin-react": "^4.2.1",
18     "vite": "^5.1.4"
19   }
20 }
```

4. Sicherheit und Konsistenz Zusätzlich erstellt npm bei jeder Installation eine `package-lock.json`. Diese Datei speichert den exakten „Fingerabdruck“ jeder installierten Bibliothek. Dadurch wird garantiert, dass das Projekt auch nach Monaten oder Jahren noch exakt so gebaut werden kann wie am Tag der Erstellung, ohne dass neuere, inkompatible Versionen von Drittanbietern den Code unbrauchbar machen.

4.3.2 JAX-RS

JAX-RS ist eine Java-API zur Erstellung von RESTful Webservices, die die Implementierung von serverseitigen HTTP-Schnittstellen stark vereinfacht. Mit JAX-RS können Entwickler HTTP-Anfragen wie GET, POST, PUT oder DELETE direkt in Java-Methoden abbilden, ohne manuell Header oder Request-Parsing behandeln zu müssen. Es unterstützt die automatische Konvertierung von Java-Objekten in Formate wie JSON oder XML und umgekehrt, wodurch die Kommunikation zwischen Client und Server effizient gestaltet wird. Typische Einsatzgebiete sind Microservices, Web-APIs oder Integrationen zwischen unterschiedlichen Systemen.

4.3.3 JSON-B

JSON-B (JSON Binding) ist eine API, die den Datenaustausch zwischen Java-Anwendungen und JSON-Daten standardisiert. Sie erlaubt das automatische Serialisieren (Konvertieren von Java-Objekten zu JSON) und Deserialisieren (JSON zu Java-Objekten) ohne manuelle Parsing-Logik. JSON-B unterstützt dabei auch komplexe Datentypen wie Listen, Maps oder verschachtelte Objekte. Durch die Verwendung von Annotations können Entwickler steuern, welche Felder serialisiert werden, wie Datumsformate behandelt werden und welche Namenskonventionen verwendet werden. JSON-B wird häufig in Verbindung mit REST-APIs genutzt, um Daten sauber zwischen Backend und Frontend auszutauschen.

4.3.4 Hibernate

Hibernate ist ein ORM-Framework (Object-Relational Mapping) für Java, das die Persistenzschicht einer Anwendung vereinfacht. Statt SQL-Abfragen manuell zu schreiben, können Entwickler Java-Objekte direkt in Datenbanktabellen abbilden. Hibernate übernimmt das Mapping zwischen Objekten und relationalen Datenbanken, einschließlich Vererbung, Beziehungen zwischen Entitäten (z.B. One-to-Many, Many-to-Many) und Transaktionsmanagement. Es bietet Funktionen wie Caching, Lazy Loading und Query Language (HQL), die die Performance optimieren und komplexe Datenzugriffe erleichtern. Hibernate wird oft in Unternehmensanwendungen, Webprojekten oder Microservices eingesetzt, bei denen eine saubere Trennung zwischen Business-Logik und Datenbankzugriff wichtig ist.

4.3.5 jose4j

jose4j ist eine Java-Bibliothek zur Umsetzung der JOSE-Standards (JSON Object Signing and Encryption). Sie bietet eine zuverlässige Möglichkeit, Daten sicher zu signieren, zu verschlüsseln und zu verifizieren. Besonders relevant ist die Bibliothek bei der Arbeit mit JSON Web Tokens (JWT), die in modernen Authentifizierungs- und Autorisierungssystemen weit verbreitet sind. Mit jose4j können Entwickler sicherstellen, dass Informationen wie Benutzeridentitäten oder Claims manipulationssicher übertragen werden, und dass nur autorisierte Parteien auf verschlüsselte Inhalte zugreifen können.

4.3.6 jBCrypt

jBCrypt ist eine Java-Bibliothek, die den BCrypt-Algorithmus zur sicheren Passwortspeicherung implementiert. Durch das Hashen von Passwörtern wird verhindert, dass sensible Daten im

Klartext in Datenbanken gespeichert werden. BCrypt verwendet Salt-Werte und adaptive Work-Factor-Parameter, wodurch selbst bei identischen Passwörtern unterschiedliche Hashes erzeugt werden. Dies erschwert Angriffe wie Rainbow-Table-Attacken oder Brute-Force-Versuche erheblich. jBCrypt wird vor allem in Webanwendungen, mobilen Apps und Backend-Systemen eingesetzt, bei denen Benutzeranmeldungen und Sicherheit eine zentrale Rolle spielen.

4.3.7 Jakarta Mail

Jakarta Mail ist eine Java-API zum Senden, Empfangen und Verwalten von E-Mails innerhalb von Java-Anwendungen. Sie unterstützt gängige Protokolle wie SMTP, IMAP und POP3 und bietet Funktionen wie Anhänge, HTML-E-Mails oder Authentifizierung. Entwickler nutzen Jakarta Mail häufig, um automatisierte Kommunikationsprozesse zu implementieren, z.B. Versenden von Registrierungs-, Passwort-Reset- oder Bestätigungs-E-Mails. Durch die API kann die gesamte E-Mail-Logik serverseitig abgebildet werden, ohne externe Tools oder Libraries einbinden zu müssen.

4.3.8 Capacitor Barcode Scanner

Der Capacitor Barcode Scanner ist ein Plugin für das Capacitor-Framework. Damit kann man direkt auf die Gerätekamera zugreifen und Barcodes oder QR-Codes scannen. Unterstützt werden dabei Formate wie QR-Codes, Code 128 und EAN. Das Plugin übernimmt die Arbeit die entstehen würde wenn wir für sowohl iOS und Android entwickeln würde, ähnlich wie Ionic generell, wodurch man als Entwickler nur eine einzige Schnittstelle braucht und nicht für jede Plattform extra Code schreiben muss. Nach dem Scannen bekommt man ein Objekt zurück, das sowohl den gescannten String als auch das erkannte Format enthält. Einbinden lässt sich das Ganze mit wenigen Zeilen Code in ein bestehendes Ionic-Projekt. In unserem Projekt 'TicketShip' nutzen wir das Plugin in der Kontrolleur-App, um die QR-Codes auf den Tickets zu scannen. Den gescannten Inhalt schicken wir dann ans Backend, das die Ticketdaten überprüft. Danach zeigt die App sowohl die Ticketinformationen als auch den Gültigkeitsstatus an.

4.3.9 JSON Web Tokens

JSON web Token kurz JWT ist ein Standard zur Sicheren Übertragung von Informationen. Ein JWT besteht aus den Teilen Header, Payload und Signature, die Base64-kodiert sind.

Der Header enthält den Algorithmus, mit dem das Token signiert wurde. Der Payload enthält die eigentlichen Daten, also zum Beispiel die User-ID oder die Rolle eines Nutzers. Die Signature

wird aus Header und Payload asymmetrisch berechnet und sie stellt sicher, dass der Token nicht manipuliert wurde.

Im Normalfall meldet sich der Benutzer an das Backend erstellt einen JWT und schickt ihn zurück. Bei jedem darauf folgenden Request schickt der Client den Token im Authorization-Header mit. Das Backend prüft dann die Signatur und bestätigt den Nutzer ohne Datenbankabfrage. JWT ist zustandslos, das heißt der Server muss keine Session speichern.

[5]

5 Planung und Realisierung

5.1 Projektorganisation

Die Projektorganisation ist ein Kernfaktor in jedem Softwareentwicklungsprojekt. Die Art der Organisation kann über Erfolg oder Misserfolg entscheiden. Der wichtigste Punkt dabei ist es, eine klare Aufgabenverteilung zu definieren, um mögliche Missverständnisse zu verhindern. Ebenso wichtig war es, auf alle Risiken vorbereitet zu sein, die während der Laufzeit des Projekts auftreten können. Die Kommunikation mit dem Auftraggeber spielte dabei eine entscheidende Rolle. Projektstatus-Meetings in der Schule, Rundgang durch das besagte Schiff und Einführung in die Geschichte der ÖGEG trug ebenfalls zum Erfolg des Projekts bei. Zusammengefasst lässt sich sagen, dass eine gute Projektorganisation wesentlich zum Erfolg eines Projekts beiträgt.

Die wöchentlichen Besprechungen mit Prof. Wokatsch-Ratzberger gaben uns den nötigen Input zur Durchführung des Projekts. Er motivierte uns, um stets am Projekt zu arbeiten und löste eventuelle Fragen, die während der verschiedenen Entwicklungsphasen auftraten. Durch sein fachliches Feedback konnten wir technische Hürden schneller überwinden und den Fokus auf die wesentlichen Projektziele beibehalten.

Besonders wertvoll war dabei der regelmäßige Austausch über den aktuellen Fortschritt, da wir so frühzeitig auf Fehlentwicklungen reagieren konnten. Diese Termine dienten uns nicht nur als fachliche Stütze, sondern auch als wichtiger Fixpunkt in der Zeitplanung, um die gesetzten Meilensteine konsequent zu verfolgen. Die genaue Aufteilung, wer welche Verantwortlichkeiten innerhalb des Teams übernommen hat, ist detailliert in der Aufgabenverteilung (siehe Kapitel 9) dokumentiert.

Zusätzlich hatten wir auch, besonders in der Endphase des Projekts, regelmäßige mit dem Auftraggeber um den derzeitigen Stand zu präsentieren und um zu notieren, wo es Verbesserungsmöglichkeiten gibt. Das hat dem ’TicketShip’ - Team sehr geholfen, indem klar definiert war, was zu erledigen ist und woran man noch arbeiten muss.

Wichtige Dokumente wie der Projektstrukturplan (PSP) und ein Gantt-Plan wurden im Rahmen unserer Diplomarbeit ausgearbeitet. Der Gantt-Plan ist ein Werkzeug, welches für die Zeitschätzung verwendet wird um diese in einem übersichtlichen Balken-Diagramm darzustellen.

5.2 Projektstrukturplan

5.2.1 Erklärung

Der Projektstrukturplan (PSP) ist das „Fundament“ eines Projekts. Er zerlegt das gesamte Projekt hierarchisch in überschaubare Teilaufgaben und Arbeitspakete. Dies ermöglicht eine einfache und klare Zuteilung der Arbeitspakete im Team. In unserem Projekt wurde der Plan in die vier Hauptbereiche Backend, Onlineshop, Backoffice und Kontrollapp aufgeteilt die auf drei Mitglieder verteilt wurden.

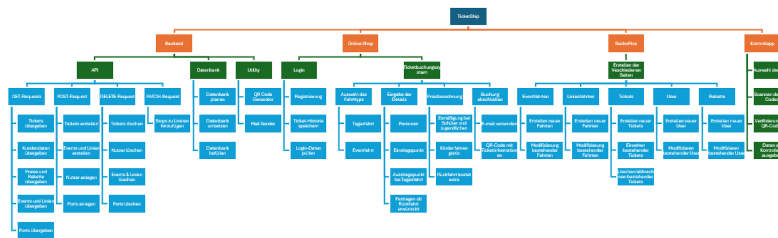

Abbildung 4: Projektstrukturplan TicketShip

Um die Grafik visuell besser darzustellen wurden die Hauptaufgaben in Orange gefärbt und die dazugehörigen Subaufgabenbereiche in grün. Die Arbeitspakete sind mit hellblau markiert.

5.3 Gantt-Plan

5.3.1 Erklärung

Ein Gantt-Plan (oder Gantt-Diagramm) ist ein Werkzeug zur Projektplanung und -steuerung. Er stellt Aufgaben eines Projekts zeitlich auf einer horizontalen Achse dar.

Dabei werden:

- Aufgaben als Balken angezeigt
- die Dauer jeder Aufgabe sichtbar gemacht
- Start- und Endzeitpunkte festgelegt

- Abhängigkeiten zwischen Aufgaben dargestellt

So kann man schnell erkennen, wann welche Aufgabe beginnt, wie lange sie dauert und ob sich Aufgaben überschneiden. Gantt-Pläne helfen dabei, Projekte übersichtlich zu planen und den Fortschritt zu kontrollieren.

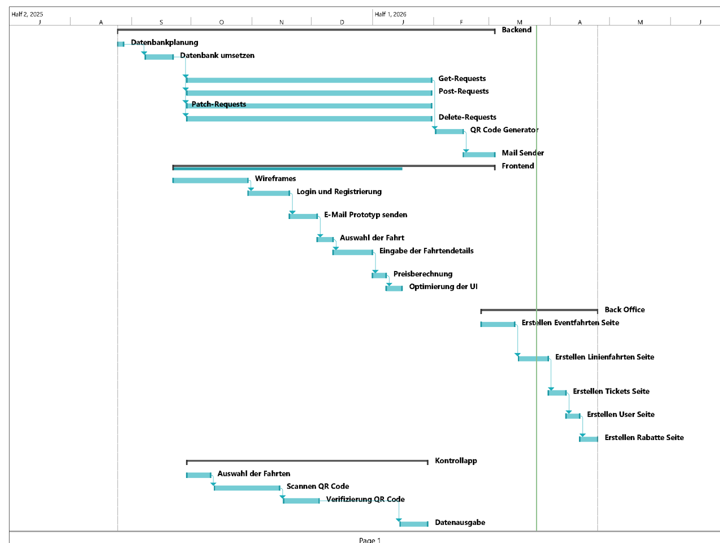

Abbildung 5: Gantt Plan TicketShip

6 Implementierung

6.1 Technischer Überblick

Das Projekt besteht aus vier Säulen(siehe Abbildung 6). Das Backend ist das Herzstück und versorgt alle anderen Applikationen über eine REST-API mit Daten und verwaltet die Daten in einer Datenbank. Der Onlineshop gibt den Kunden die Möglichkeit, ihre Tickets im Vorhinein online zu erwerben. Im Admin Panel können Administratoren die Tickets und Fahrten verwalten und erhalten Statistiken zu den jeweiligen Fahrten. Mit der Kontrollapp können die Tickets schnell und einfach auf ihre Gültigkeit kontrolliert werden.

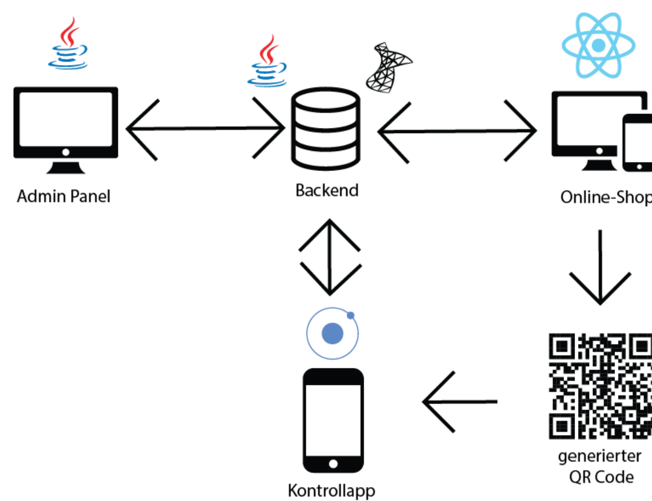

Abbildung 6: Aufbau TicketShip

6.2 Backend

Das Backend versorgt das Adminpanel, den Onlineshop und die Kontrollapp mit den notwendigen Daten und persistiert diese in der Datenbank. Außerdem beinhaltet das Backend drei Utillity Klassen.

6.2.1 Utilities

MailSender

MailSender ist eine Klasse, die die Verbindung zum SMTP (Simple Mail Transfer Protocol) Server herstellt und die Email ansprechend mit HTML formatiert. Diese baut auf der Jakarta Mail Bibliothek und übernimmt den gesamten automatisierten Emailversand.

QRCodeGenerator

Diese Klasse erstellt mithilfe der Google zxing (Zebra Crossing) Bibliothek den QR Code, der mit dem MailSender an die Kunden übermittelt wird.

Listing 3: QR Code Generator

```
1       public static Path generateQRCode(String text, String fileName) throws Exception {
2           BitMatrix matrix = new MultiFormatWriter()
3                   .encode(text, BarcodeFormat.QR_CODE, 300, 300);
4
5           Path path = Path.of(fileName);
6           MatrixToImageWriter.writeToPath(matrix, "PNG", path);
7           return path;
8       }
```

Der QR Code wird in einem temporären Ordner zwischengespeichert für die direkte Weiterverarbeitung im MailSender

JWTGen

Übernimmt mit der jose4j Bibliothek die Erstellung und Validierung von JWT Tokens.

Listing 4: JWT Generator

```
1       public String generateToken(long userId) {
2           Users user = usersDAO.getUserById(userId);
3
4           Jsonb jsonb = JsonbBuilder.create();
5           String json = jsonb.toJson(user);
6
7           JsonWebEncryption jwe = new JsonWebEncryption();
8           jwe.setPayload(json);
9           jwe.setAlgorithmHeaderValue(KeyManagementAlgorithmIdentifiers.A128KW);
10          jwe.setEncryptionMethodHeaderParameter(
11                  ContentEncryptionAlgorithmIdentifiers.AES_128_CBC_HMAC_SHA_256
```

```
12          );
13          jwe.setKey(key);
14
15          try {
16              return jwe.getCompactSerialization();
17          } catch (JoseException e) {
18              throw new RuntimeException(e);
19          }
20      }
```

Im Token werden Daten vom angemeldeten Nutzer gespeichert die notwendig sind um die Verifizierung des Nutzers durchzuführen und dessen Berechtigungen einzusehen.

Listing 5: JWT Verifizerer

```
1       public boolean validateToken(String token){
2           JsonWebEncryption jwe = new JsonWebEncryption();
3           try {
4               jwe.setCompactSerialization(token);
5               jwe.setKey(key);
6               String payload = jwe.getPayload();
7               Jsonb jsonb = JsonbBuilder.create();
8               Users user = jsonb.fromJson(payload, Users.class);
9               return usersDAO.getUserById(user.getId()) != null;
10          } catch (JoseException e) {
11              return false;
12          }
13      }
```

Im Validierungsprozess werden dann die gespeicherten Daten überprüft und dementsprechend der Zugriff erlaubt oder verweigert

6.2.2 API

Die API versorgt alle Komponenten mit den notwendigen Daten, speichert die Daten strukturiert und versendet die Tickets über Emails an die Kunden

HTTP-Methoden

Das Hypertext Transfer Protocol (HTTP) ist ein grundlegendes Protokoll für die Kommunikaton. Es definiert verschiedene Methoden, mit denen Aktionen auf einem Webserver ausgeführt werden können.

In unserem Projekt kommen die Methoden GET, POST, PATCH und DELETE zum Einsatz.

Endpunkte

- Lines
  - GET getAllLines -> übergibt alle existierenden Linien

  - POST addLine -> fügt eine Linie hinzu
  - PATCH addStop -> Fügt einen Stop zu einer Linie hinzu
  - GET getStopsByLine -> übergibt alle Stops einer Linie
  - GET getPassengersForStop -> übergibt, wie viele Passagiere pro Stop ein- und aussteigen

- Linientickets
  - POST createLineTicket -> erstellt ein Linienticket und übermittelt es mit dem MailSender an die Kunden
  - GET checkTicket -> überprüft ob das Ticket existiert
  - PUT validateTicket -> entfernt die Angegebene Anzahl an Kunden vom Ticket
  - DELETE deleteTicket -> löscht ein Ticket
  - GET getTicketsByLine -> gibt alle Tickets einer Linie zurück
  - GET calculatePrice -> berechnet den Preis der Tickets
- Events
  - GET getAllEvents -> übergibt alle existierenden Events
  - POST addEvent -> fügt ein Event hinzu
  - GET getSeatsForEvent -> übergibt die Sitzplätze für Events
  - GET addSeat -> reserviert einen Sitzplatz
- Eventtickets
  - POST createEventTicket -> erstellt ein Eventticket und übermittelt es mit dem MailSender an die Kunden
  - GET checkTicket -> überprüft, ob das Ticket existiert
  - PUT validateTicket -> entfernt die Angegebene Anzahl an Kunden vom Ticket
  - DELETE deleteTicket -> löscht ein Ticket
  - GET getTicketsByEvent -> gibt alle Tickets eines Events zurück
  - GET calculatePrice -> berechnet den Preis der Tickets
- Users

- GET login -> überprüft, ob der User existiert und gibt ein JWT Token zurück
- POST register -> legt einen User an

6.2.3 Datenbank

Die Datenbank ist das Gehirn unserer Diplomarbeit. Sie speichert alle Nutzer, Tickets, Linien, Events, Preise und Rabatte. Wir haben uns für MySQL entschieden, welche bekannt für ihre starken Sicherheits-Features und der großen Entwickler-Community [6] ist. Des Weiteren ist MySQL open-source, was besonders aus der Finanzsicht perfekt für den Verein ist.

Datenbank Schema

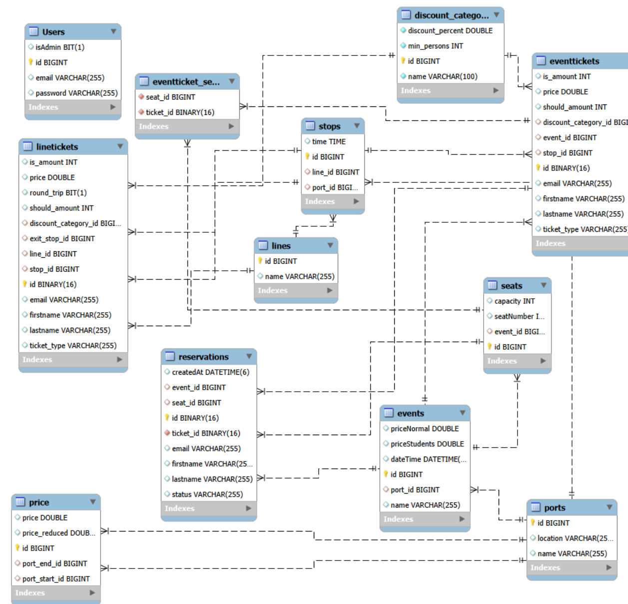

Abbildung 7: Schema Datenbank

Benutzerverwaltung und Preisregeln

Tabelle: Users

Diese Tabelle bildet die Grundlage für die Authentifizierung. Neben den Login-Daten (E-Mail und Passwort) wird über das Feld `isAdmin` gesteuert, ob ein Nutzer nur Buchungen vornehmen darf oder Zugriff auf das Management-Backend (z.B. zum Anlegen von Linien oder Preisen) hat.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
Users	id	BIGINT (PK)	Interner Primärschlüssel.
	email	VARCHAR(255)	Eindeutige Adresse; dient als Benutzername.
	password	VARCHAR(255)	Speichert das Passwort (üblicherweise gehasht).
	isAdmin	BIT(1)	Berechtigungssteuerung (Admin vs. Kunde).

Tabelle: discount_categories

Hier wird die Geschäftslogik für Rabatte hinterlegt. Das System unterstützt volumenbasierte

Rabatte: Wenn eine Buchung die Anzahl `min_persons` erreicht, kann der `discount_percent` angewendet werden. Dies ist sowohl für Event- als auch für Linientickets relevant.

discount_categories	id	BIGINT (PK)	Primärschlüssel für Rabattgruppen.
	name	VARCHAR(100)	Name (z.B. "Gruppenrabatt", SSchüler").
	discount_percent	DOUBLE	Prozentualer Abzug vom Normalpreis.
	min_persons	INT	Schwellenwert für die Anwendung des Rabatts.

Infrastruktur und Linienbetrieb

Tabelle: lines, ports und stops

Diese drei Tabellen bilden das logische Rückgrat des Verkehrsnetzes. Eine Line ist eine abstrakte Route (z.B. "Muttertagsfahrt nach Aschach via Ottensheim"). Ein Port ist ein physischer Ort. Die Tabelle Stops ist eine Verknüpfungstabelle, die definiert, welche Linie wann an welcher Anlegestelle hält. Dies ermöglicht einen dynamischen Fahrplan.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
lines	id	BIGINT (PK)	Identifikator für eine Linienfahrt.
	name	VARCHAR(255)	Bezeichnung der Linie.
ports	id	BIGINT (PK)	Identifikator für einen Anlegeort.
	name	VARCHAR(255)	Anzeigename des Hafens.
	location	VARCHAR(255)	Geografische Zusatzinfos oder Adresse.
stops	id	BIGINT (PK)	Einzeleintrag im Fahrplan.
	time	TIME	Die Uhrzeit, zu der dieser Halt stattfindet.
	line_id	BIGINT (FK)	Verweist auf die zugehörige Linie.
	port_id	BIGINT (FK)	Verweist auf den Ort des Halts.

Tabelle: price

Das Preissystem arbeitet streckenabhängig. Statt eines Fixpreises pro Linie erlaubt diese Tabelle die Definition von Preisen zwischen einem `port_start_id` und einem `port_end_id`. So können Teilstrecken unterschiedlich bepreist werden.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
price	id	BIGINT (PK)	Eindeutiger Identifikator.
	price	DOUBLE	Der Standardpreis für diesen Streckenabschnitt.
	price_reduced	DOUBLE	Ermäßigung für berechtigte Personen.
	port_start_id	BIGINT (FK)	Referenz auf den Abfahrtshafen.
	port_end_id	BIGINT (FK)	Referenz auf den Zielhafen.

Event-Management und Kapazitäten

Tabelle: events und seats

Events sind zeitlich begrenzte Sonderveranstaltungen (z.B. "Martiniganslessen"). Ein Event findet an einem bestimmten `port_id` statt. Die Tabelle Seats verwaltet die verfügbaren Plätze pro Event. Dies ermöglicht eine sitzplatzgenaue Buchung und verhindert Überbuchungen durch das Feld `capacity`.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
events	id	BIGINT (PK)	Identifikator für das Event.
	name	VARCHAR(255)	Titel des Ereignisses.
	dateTime	DATETIME	Exaktes Datum und Uhrzeit.
	port_id	BIGINT (FK)	Veranstaltungsort.
	priceNormal	DOUBLE	Preis für Erwachsene/Normalzahler.
	priceStudents	DOUBLE	Rabattierter Preis für Studenten.
seats	id	BIGINT (PK)	Identifikator für einen Platz.
	seatNumber	INT	Physische Nummerierung des Platzes.
	capacity	INT	Anzahl der Personen pro SSitz"(oft 1).
	event_id	BIGINT (FK)	Zuordnung zu einem speziellen Event.

Buchungstransaktionen

Tabelle: linetickets und eventtickets

Das System trennt Tickets für den Linienverkehr von Event-Tickets. Linetickets speichern Streckeninformationen (`stop_id` bis `exit_stop_id`) und ob es sich um eine Rückfahrt handelt. Eventtickets sind an ein spezifisches Event gebunden. Beide Tabellen nutzen `BINARY(16)` (UUIDs) als Primärschlüssel, um eine weltweit eindeutige Identifizierung (z.B. für QR-Code-Scanner) zu gewährleisten.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
line tickets	id	BINARY(16)	Eindeutige Ticket-ID (UUID).
	firstname / lastname	VARCHAR(255)	Personalisierung des Tickets.
	price	DOUBLE	Der beim Kauf tatsächlich gezahlte Preis.
	round_trip	BIT(1)	Flag: Beinhaltet das Ticket eine Rückfahrt?
	line_id	BIGINT (FK)	Referenz auf die genutzte Linie.
	stop_id / exit_stop	BIGINT (FK)	Einstiegs- und Ausstiegspunkt.
event tickets	id	BINARY(16)	Eindeutige Ticket-ID (UUID).
	event_id	BIGINT (FK)	Zuordnung zum Event.
	discount_category_id	BIGINT (FK)	Referenz auf genutzte Rabattregel.
	is_amount	INT	Anzahl der gebuchten Personen/Plätze.
	email	VARCHAR(255)	Empfängeradresse für den Ticketversand.

Tabelle: reservations

Die Reservierungstabelle dient als Zwischenspeicher. Wenn ein Nutzer Plätze auswählt, werden diese hier mit einem `status` (z.B. "PENDING") blockiert. Erst nach erfolgreicher Zahlung wird die `ticket_id` gesetzt und der Prozess abgeschlossen. Das Feld `createdAt` ermöglicht es, abgelaufene Reservierungen automatisch freizugeben.

Tabelle	Attribut	Datentyp	Funktionelle Beschreibung
reser vations	id	BINARY(16)	Identifikator der Reservierung.
	status	VARCHAR(255)	Status-Workflow (Offen, Bezahlt, Storno).
	createdAt	DATETIME(6)	Zeitstempel für Timeout-Logik.
	event_id	BIGINT (FK)	Verknüpfung zum reservierten Event.
	seat_id	BIGINT (FK)	Verknüpfung zum reservierten Sitz.
	ticket_id	BINARY(16) (FK)	Referenz auf das finale Ticket.
	email / name	VARCHAR(255)	Temporäre Speicherung der Kundendaten.

6.3 Frontend

6.3.1 Login- und Registrierungs-Page

Einleitung

Ziel

Der Benutzer muss ein Konto erstellen, um die Buchungen bzw. Fahrten zu kaufen und zu verwalten.

Überblick der Pages

- Es wurde eine Login-Page erstellt, um sich mit einem bereits existierenden Konto anzumelden.
- Wenn noch kein Konto erstellt wurde, kann über einen Button in der Login-Page auf eine Registrier-Page navigiert werden.

Planung & Anforderungen

Funktionale Anforderungen

- Benutzer kann sich ein Konto bei der ÖGEG anlegen.
- Benutzer kann sich anmelden.
- Benutzer kann seine gekauften Ticketdaten einsehen

Nicht-funktionale Anforderungen

- Einfache Bedienung.
- Sichere Datenverwaltung.
- Klare Fehlermeldungen:
  - Anmeldung fehlgeschlagen: E-Mail / Passwort falsch.
  - Konto mit dieser E-Mail existiert bereits (bei Versuch, einen Account mit einer bereits vorhandenen E-Mail zu erstellen).

Design

Farbpalette

Als Farbpalette wurde die schlichte Palette bestehend aus Grau, Weiß und Dunkelrot verwendet, die bereits auf der Website sichtbar ist.

Screenshots

Abbildung 8: Login-Page

Abbildung 9: Registrier-Page

Navigation

Benutzer sieht zuerst die Login-Page (siehe Grafik Login-Page 8), von dieser Seite aus hat er mehrere Möglichkeiten:

- Login-Daten eingeben und sich anmelden
- Neues Konto anlegen über Registrier-Page (siehe Grafik Registrierungs-Page 9)

Weitere Navigationsmöglichkeiten:

- Auf der Registrier-Page kann der Benutzer zurück zur Login-Page navigieren.

Technische Umsetzung

Login-Page

- Eingabefelder:
  - E-Mail-Adresse
  - Passwort
- Button zum Anmelden
- Verlinkungen zu:
  - Neues Konto anlegen
- Backend-Funktion:
  - Überprüfung der Zugangsdaten in der Datenbank

Registrierungs-Page

- Eingabefelder:
  - Vor- und Nachname
  - E-Mail-Adresse
  - Passwort
  - Passwort bestätigen
- Button zum Anlegen des Kontos
- Verlinkung zur Login-Page

- Backend-Funktion:
  - Neuen Benutzer in der Datenbank anlegen

Testfälle

- Falsche Logindaten: Fehlermeldung
- Korrekte Logindaten: Weiterleitung auf Home-Page
- Erfolgreiche Registrierung: Weiterleitung auf Home-Page

Login-Logik und Datenverwaltung

Damit die App weiß, welcher Nutzer gerade angemeldet ist, nutzt sie ein modernes Verfahren mit sogenannten JWT-Tokens und einen kleinen Trick im Speicher des Browsers.

Das JWT-Token (Der digitale Ausweis) Wenn der Login erfolgreich ist, schickt der Server ein verschlüsseltes Paket zurück: das JWT (JSON Web Token). Man kann sich das wie einen digitalen Besucherausweis vorstellen. Dieser wird im sessionStorage abgelegt. Das ist sicher, weil die Daten automatisch gelöscht werden, sobald der Nutzer den Tab im Browser schließt.

Listing 6: Speicherung des JWT-Tokens

```
1       const data = await response.json()
2           if (data.token) {
3       sessionStorage.setItem('token', data.token)
4   }
```

Bessere Fehlermeldungen durch den LocalStorage Normalerweise sagt ein Server aus Sicherheitsgründen oft nur: „Anmeldung fehlgeschlagen“. Das ist für den Nutzer aber nervig, wenn er nicht weiß, ob er sich bei der E-Mail vertippt hat oder nur das Passwort falsch ist.

Deshalb haben wir eine zusätzliche Logik eingebaut:

Wenn der Server den Zugriff verweigert, schaut die App kurz im localStorage nach, ob diese E-Mail-Adresse dort schon einmal gespeichert wurde (z. B. durch eine vorherige Registrierung).

Ist die E-Mail bekannt, zeigt die App: „Falsches Passwort“.

Ist die E-Mail völlig unbekannt, zeigt sie: „Kein Konto gefunden“.

Das macht die Bedienung der App viel freundlicher, da der Nutzer sofort weiß, wo der Fehler liegt. Sobald der Login klappt, werden auch Name und E-Mail im sessionStorage hinterlegt, damit die App den Nutzer auf der Startseite direkt persönlich begrüßen kann.

Listing 7: Speicherung des JWT-Tokens

```
1       const localUsers = JSON.parse(localStorage.getItem('users') || '[]')
2       const localUser = localUsers.find(u => u.email === email)
3
4       sessionStorage.setItem('user', 'eingeloggt')
5       sessionStorage.setItem('email', email)
6       sessionStorage.setItem('firstname', localUser?.firstname || '')
7       sessionStorage.setItem('lastname', localUser?.lastname || '')
8
9       navigate('/')
```

Registrierungs-Logik

Damit ein neuer Nutzer überhaupt Teil des Systems werden kann, muss er sich zuerst registrieren. Das passiert über eine Schnittstelle (API), an die wir die Daten sicher schicken.

Datenübertragung an das Backend Wenn der Nutzer auf „Registrieren“ klickt, nimmt die App die E-Mail und das Passwort und bereitet sie für den Versand vor. Wichtig ist hierbei, dass die E-Mail mit .trim().toLowerCase() bereinigt wird. So verhindern wir Fehler durch versehentliche Leerzeichen oder Großbuchstaben. Die Daten werden dann als JSON-Paket per POST-Anfrage an den Server gesendet.

Listing 8: Registrierung eines neuen Nutzers

```
1       try {
2           const response = await fetch(API, {
3               method: 'POST',
4               headers: { 'Content-Type': 'application/json' },
5               body: JSON.stringify({
6                   email: email.trim().toLowerCase(),
7                   password
8               })
9           })
```

Speicherung für die Benutzererfahrung Sobald der Server grünes Licht gibt und das Konto erstellt hat, speichern wir die E-Mail und den Namen zusätzlich im localStorage.

Das hat einen einfachen Grund: Wie schon bei der Login-Logik erwähnt, hilft uns das später dabei, dem Nutzer bessere Fehlermeldungen anzuzeigen. Wenn er sich das nächste Mal einloggen will, „erinnert“ sich der Browser daran, dass diese E-Mail bei uns schon bekannt ist. Das macht die ganze App ein Stück weit intelligenter und nutzerfreundlicher.

6.3.2 Ticketbuchung

Einleitung

Ziel

Der Benutzer soll Tickets einfach und direkt über die Website buchen können, ohne den bisherigen telefonischen Prozess nutzen zu müssen.

Überblick des Buchungssystems

- Es wurde ein Webshop implementiert, über den Tickets online gebucht werden können.
- Der Buchungsprozess wird über einen Button gestartet.
- Der Benutzer wird schrittweise durch den gesamten Buchungsvorgang geführt.

- Am Ende erhält der Benutzer eine Übersicht und kann die Buchung abschließen.

Planung & Anforderungen

Funktionale Anforderungen

- Benutzer kann zwischen verschiedenen Fahrten wählen (Tagesfahrt oder Event-Fahrt).
- Benutzer kann Ticketdaten eingeben (Datum, Personen, Strecke).
- Benutzer kann persönliche Daten eingeben.
- System berechnet automatisch den Preis.
- Benutzer kann Buchung abschließen.

Nicht-funktionale Anforderungen

- Einfache und verständliche Bedienung.
- Schnelle Ladezeiten.
- Sichere Verarbeitung der Benutzerdaten.
- Klare Anzeige von Fehlern und Eingabefehlern.

Design

Buchungsablauf

Der Benutzer startet den Buchungsprozess über einen Button und durchläuft folgende Schritte:

- Auswahl der Fahrt (siehe Grafik Fahrtenauswahl 10):
  - Tagesfahrt oder Event-Fahrt
- Eingabe der Ticketdaten (siehe Grafik Eingabe der Details 11):
  - Datum
  - Anzahl der Personen (Erwachsene, Jugendliche, Kinder)
  - Auswahl der Fahrt
  - Hin- und Rückfahrt oder nur Hinfahrt
- Eingabe der persönlichen Daten (siehe Grafik Eingabe persönlicher Daten 12):

  - Vor- und Nachname
  - E-Mail-Adresse
- Übersicht:
  - Anzeige aller eingegebenen Daten
  - Preisberechnung
  - Möglichkeit zur Korrektur
- Abschluss (siehe Grafik Zusammenfassung der Daten 13):
  - Buchung wird kostenpflichtig durchgeführt

Navigation

- Benutzer startet auf der Hauptseite und gelangt über einen Button zur Ticketbuchung.
- Während des Buchungsprozesses kann der Benutzer:
  - Schritte zurückgehen
  - Eingaben ändern
- Nach erfolgreicher Buchung wird der Benutzer zu einer Bestätigungsseite weitergeleitet.

Screenshots

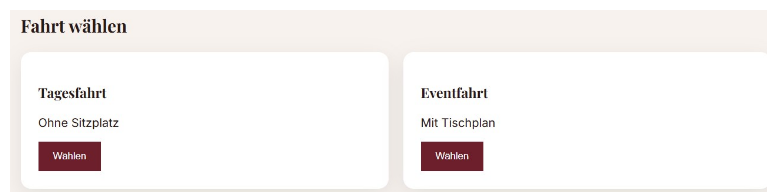

Abbildung 10: Fahrtenauswahl

Abbildung 11: Eingabe der Details

Abbildung 12: Eingabe persönlicher Daten

Abbildung 13: Zusammenfassung der Eingaben und Preisberechnung

Technische Umsetzung

Buchungsstart

- Button zum Starten des Buchungsprozesses
- Weiterleitung zur Auswahl der Fahrt

Ticketdaten-Page

- Eingabefelder:
  - Datum
  - Personenanzahl
  - Fahrt auswählen
  - Hin- und Rückfahrt
- Backend-Funktion:
  - Speicherung der eingegebenen Daten

Persönliche Daten-Page

- Eingabefelder:
  - Vor- und Nachname
  - E-Mail-Adresse
- Backend-Funktion:
  - Speicherung der Kundendaten

Übersichts-Page

- Anzeige aller Buchungsdaten
- Berechnung des Gesamtpreises
- Button zum Abschließen der Buchung
- Backend-Funktion:
  - Validierung der Daten
  - Durchführung der Buchung

Testfälle

- Fehlende Eingaben: Fehlermeldung wird angezeigt
- Ungültige Daten: Benutzer muss Eingaben korrigieren
- Korrekte Eingaben: Buchung wird erfolgreich durchgeführt

- Preisberechnung korrekt
- Weiterleitung zur Bestätigungsseite nach erfolgreicher Buchung

Buchungs-Logik und Sitzplatzwahl

Die Buchungs-Logik sorgt dafür, dass die verfügbaren Events und die dazugehörigen Sitzplätze aktuell im Frontend angezeigt werden. Dieser Prozess wird eingeleitet, sobald die Seite geladen oder ein Event ausgewählt wird.

Automatisches Laden der Event-Liste: Beim Öffnen der Buchungsseite wird durch den useEffect-Hook sofort eine Anfrage an das Backend gesendet. Das Ziel ist es, alle aktuell verfügbaren Eventfahrten abzurufen. Sobald die Antwort vom Backend kommt, werden diese Daten dem Nutzer in einer Auswahl-Liste zur Verfügung gestellt. Sollte die Verbindung zum Server unterbrochen sein, wird ein Fehler in der Konsole ausgegeben.

Listing 9: Holen der Event-Liste beim Drücken des Buchungsbuttons

```
1  useEffect(() => {
2      const fetchEvents = async () => {
3      try {
4          const response = await fetch("{SERVER_URL}/api/events");
5          if (response.ok) {
6              const data = await response.json();
7              setEvents(data);
8          }
9      } catch (error) {
10         console.error("Fehler beim Laden der Events:", error);
11     }
12 };
13 fetchEvents();
14 }, []);
```

Dynamische Abfrage der Sitzplätze Sobald sich der Nutzer für eine Veranstaltung entschei-det, wird die Funktion handleEventSelect ausgelöst. Diese steuert die Anzeige der Sitzplätze basierend auf der Auswahl:

- Spezifische Abfrage: Die App erkennt die ID des gewählten Events und startet mit fetchSeats eine neue Anfrage an das Backend.
- Filterung: Es werden nur die Sitzplätze geladen und angezeigt, die technisch mit der ID dieses Events verknüpft sind.
- Zurücksetzen: Wird die Auswahl aufgehoben, wird die Liste der Sitzplätze geleert.

Dies stellt sicher, dass der Nutzer immer nur die Plätze sieht, die für das aktuell gewählte Event relevant sind.

Listing 10: Sitzplaetze laden nach Event-Auswahl

```
        const fetchSeats = async (eventId) => {
            try {
            const response = await fetch({SERVER_URL}/api/events/${eventId}/seats);
        if (response.ok) {
            const data = await response.json();
            setSeats(data);
            }
        } catch (error) {
        console.error("Fehler beim Laden der Sitze:", error);
        }
};
```

Auswahl der Ein- und Ausstiegspunkte

Für Tagesfahrten wurde eine Logik implementiert, die es dem Nutzer ermöglicht, Einstiegs- und Ausstiegspunkt flexibel festzulegen. Dies ist besonders wichtig, wenn die Buchung nicht an ein festes Event mit vorgegebenen Haltestellen gebunden ist.

Die Funktion handlePortChange Um die Auswahl im State der Anwendung zu speichern, wird eine zentrale Funktion genutzt. Diese erfüllt zwei Aufgaben gleichzeitig: Sie speichert sowohl die technische ID des Hafens (für die Datenbank) als auch den Namen des Hafens (für die Anzeige in der UI).

Dabei wird der Spread-Operator (...booking) verwendet, um bestehende Buchungsdaten nicht zu überschreiben. Falls die Auswahl zurückgesetzt wird, setzt die Logik die Werte auf undefined oder einen leeren String zurück.

Listing 11: Zentrale Funktion zur Hafenwahl

```
const handlePortChange = (field, idField, e) => {
const selectedId = e.target.value;
const selectedName = e.target.options[e.target.selectedIndex].text;

setBooking({
    ...booking,
    [idField]: selectedId ? Number(selectedId) : undefined,
    [field]: selectedId ? selectedName : ""
    });
};
```

Dynamische Anzeige im Interface

Im User Interface werden die Auswahlfelder (Dropdowns) für den Einstiegs- und Ausstiegspunkt basierend auf dem Buchungstyp gesteuert:

- Einstiegspunkt: Hier kann der Nutzer aus einer Liste (ports.map) den gewünschten Starthafen wählen. Ist bereits ein festes Event ausgewählt, wird dieses Feld deaktiviert, da Start und Ziel in diesem Fall meist vordefiniert sind.
- Bedingte Anzeige des Ausstiegspunkts: Das Feld für den Ausstiegspunkt wird nur dann eingeblendet, wenn kein spezifisches Event ausgewählt wurde (!booking.eventId). Dies ermöglicht bei freien Tagesfahrten eine individuelle Routenplanung.

Listing 12: JSX-Struktur für die Hafenwahl

```

<div className="port-selection">
        <label>Einstiegspunkt</label>
        <select value={booking.startPortId || ""} onChange={(e) =>
            handlePortChange("start", "startPortId", e)} disabled={!!booking.eventId}>
          <option value="">-- Bitte Hafen wählen --</option>
          {ports.map(port => (
            <option key={port.id} value={port.id}>{port.name}</option>
          ))}
        </select>

        {!booking.eventId && (
          <>
            <label style={{ marginTop: "15px", display: "block" }}>Ausstiegspunkt</label>
            <select value={booking.endPortId || ""} onChange={(e) =>
                handlePortChange("end", "endPortId", e)}>
              <option value="">-- Bitte Hafen wählen --</option>
              {ports.map(port => (
                <option key={port.id} value={port.id}>{port.name}</option>
              ))}
            </select>
          </>
        )}
      </div>
```

Durch diese Trennung wird sichergestellt, dass die Anwendung flexibel auf verschiedene Buchungsszenarien reagiert und Fehleingaben durch das Deaktivieren von Feldern minimiert werden.

Optionen für Hin- und Rückfahrt sowie Validierung

Zusätzlich zur Hafenwahl bietet die Anwendung spezifische Optionen für individuelle Tagesfahrten an. Diese Funktionen sind durch logische Abfragen so gesteuert, dass sie nur erscheinen, wenn keine feste Veranstaltung (Event) ausgewählt wurde.

Auswahl der Rückfahrt-Option Über eine Checkbox kann der Nutzer angeben, ob eine Hin- und Rückfahrt gewünscht ist. Der Zustand dieser Auswahl wird direkt im booking-Objekt unter dem Feld roundTrip gespeichert. Da diese Option bei festen Events meist vordefiniert ist, wird das gesamte Element nur angezeigt, wenn booking.eventId leer ist.

Listing 13: Checkbox für die Rückfahrt-Option

```
{!booking.eventId && (
    <label>
    <input
        type="checkbox"
        checked={!!booking.roundTrip}
        onChange={(e) => setBooking({ ...booking, roundTrip: e.target.checked })}
    />
    <strong>Hin- & Rückfahrt erwünscht?</strong>
    </label>
)}
```

Validierung der Hafenwahl Um logische Fehler bei der Buchung zu vermeiden, wurde zur Laufzeit überprüft, ob der Einstiegs -und Ausstiegshafen identisch ist. Es ist technisch nicht sinnvoll, den

gleichen Hafen als Einstiegs- und Ausstiegspunkt zu wählen. Die App prüft daher permanent drei Bedingungen:

- Es ist kein festes Event ausgewählt.
- Die ID des Starthafens entspricht der ID des Zielhafens.
- Es wurden bereits Häfen ausgewählt (die Felder sind nicht leer).

Sind alle diese Bedingungen erfüllt, wird dem Nutzer sofort ein Warnhinweis in roter Schrift eingeblendet. Dies verhindert, dass fehlerhafte Buchungsdaten an das Backend gesendet werden können.

Listing 14: Logik zur Fehleranzeige bei identischen Häfen

```
{!booking.eventId && booking.startPortId === booking.endPortId && booking.startPortId && (

<p style={{ color: "red", fontWeight: "bold" }}>
    Start und Ziel dürfen nicht identisch sein!
</p>
)}
```

Zusammenfassung und Buchungsabschluss

Die Komponente StepSummary stellt den letzten Schritt im Buchungsprozess dar. Hier werden alle gewählten Optionen zusammengeführt, die Preise berechnet und die finale Buchung an das Backend übermittelt.

Dynamische Preisberechnung: Ein zentrales Merkmal ist die automatische Preisermittlung über einen useEffect-Hook. Dabei unterscheidet das System zwischen zwei Szenarien:

- **Eventfahrten:** Hier werden die Preise direkt aus den Objektdaten der Veranstaltung (z. B. Normalpreis oder ermäßigter Preis) ausgelesen und mit der Personenanzahl multipliziert.
- **Linienfahrten:** Da die Preise hier von der Strecke (Start- und Zielhafen) sowie von möglichen Gruppenrabatten abhängen, wird für jede Ticketkategorie (Erwachsene, Kinder, Jugendliche) eine separate Anfrage an die API gestellt.

Listing 15: Abfrage der Preise über die API

```
const fetchPrice = async (type, amount) => {
    const params = new URLSearchParams({
        startPortId: booking.startPortId,
        endPortId: booking.endPortId,
        ticketType: type,
        roundTrip: !!booking.roundTrip,
        amount: totalPersons
});
const response = await fetch(.../api/tickets/price?${params.toString()});
return await response.json();
};
```

Abschluss der Buchung und Sitzplatzreservierung Beim Klick auf den Buchungs-Button wird die Funktion handleConfirm ausgeführt. Diese übernimmt die finale Kommunikation mit dem Server in zwei Schritten:

Ticket-Erstellung: Zuerst werden die Nutzerdaten und die Reiseverbindung an die Ticket-API gesendet. Je nachdem, ob es sich um ein Event oder eine Linienfahrt handelt, wählt die App automatisch den richtigen Endpunkt (/tickets oder /event-tickets).

Optionale Reservierung: Falls der Nutzer im Vorfeld einen Tischwunsch geäußert hat (über tableId), wird nach der erfolgreichen Ticket-Erstellung automatisch eine zweite Anfrage an die Reservations-API gesendet. Dabei wird das neue Ticket direkt mit dem gewählten Sitzplatz verknüpft.

Listing 16: Finalisierung der Buchung mit optionaler Reservierung

```
1  if (res.ok) {
2      if (booking.tableId) {
3          const ticket = await res.json();
4          await fetch(".../api/reservations", {
5              method: "POST",
6              body: JSON.stringify({
7              ticketId: ticket.id,
8              seatId: booking.tableId
9          })
10     });
11 }
12 alert("Ticket erfolgreich gebucht!");
13 navigate("/");
14 }
```

Transparenz für den Nutzer: Die Oberfläche zeigt dem Nutzer eine detaillierte Auflistung der Kosten pro Kategorie (z. B. "2x Erwachsene á 15.00€"). Zudem wird ein Hinweis eingeblendet, dass Tischauswahlen lediglich als Wunsch gewertet werden. Erst wenn der Gesamtpreis erfolgreich berechnet wurde, wird der Buchungs-Button freigeschaltet, um fehlerhafte Datensätze zu vermeiden.

6.3.3 Ticketvalidierungs-App (Ionic)

Übersicht

Die Ticketvalidierungs-App ist eine App für Kontrolleure. Sie scannen damit QR-Codes auf Tickets und prüfen, ob diese gültig sind. Die App läuft auf Android- und iOS-Geräten und kommuniziert über HTTP mit dem Jakarta-EE-Backend, das die Ticketdaten verwaltet.

Es gibt zwei Phasen. Zuerst scannt der Kontrolleur einen QR-Code. Die App erkennt, ob es ein Linien- oder Veranstaltungsticket ist, und fragt die Ticketdaten vom Backend ab. In der zweiten Phase bestätigt der Kontrolleur die Entwertung. Die App überträgt die gewählte Anzahl an das Backend. Das Backend setzt den Entwertungszähler in der Datenbank hoch. Zwischen beiden Phasen kann der Kontrolleur die Zahl der Personen ändern.

Es gibt zwei Arten von Tickets: Linientickets und Veranstaltungstickets. Linientickets sind an eine bestimmte Fahrtlinie gebunden. Veranstaltungstickets haben zugewiesene Sitzplätze. Bei Linientickets prüft die App zusätzlich, ob das Ticket zur aktuell gewählten Linie passt. Stimmt die Linie nicht überein, wird das Ticket abgelehnt, bevor man es einlösen kann.

Technische Grundlage

Die App baut auf Ionic 7 mit Angular und Capacitor auf. Für den QR-Scan kommt das Plugin `@capacitor/barcode-scanner` zum Einsatz — es öffnet die Gerätekamera und gibt den gescannten Inhalt als String zurück. HTTP-Aufrufe laufen über Angulars `HttpClient`. Der `TicketService` ist für Ticketprüfung und Entwertung zuständig, der `LineService` liefert Linien und Veranstaltungen für die Übersicht.

DetailComponent

Der `DetailComponent` macht den Scan und die Entwertung und hält den Zustand dazwischen. Die Komponente wird über eine Route mit dem Parameter `id` aufgerufen. Beginnt die ID mit `event-`, ist es eine Veranstaltungssession, sonst eine Liniensession. Das wird einmal in `ngOnInit` festgelegt und bestimmt, welche API-Endpunkte die App anspricht.

startScan Die Methode `startScan` öffnet die Kamera über den Capacitor Barcode Scanner und wartet auf ein Scan-Ergebnis. Zuerst wird `successInfo` zurückgesetzt, damit keine Daten vom letzten Durchlauf stehenbleiben.

Der gescannte String wird als URL behandelt. Die Methode splittet ihn an den Schrägstrichen und nimmt das letzte Segment. Ist das Segment leer, bricht sie ab und setzt `boxColor` auf `'red'`.

Das erste Zeichen des Segments bestimmt den Ticket-Typ. Ein `'E'` heißt Veranstaltungsticket — das Präfix wird abgeschnitten und `checkEventTicket` aufgerufen. Ein `'L'` heißt Linienticket — gleiches Vorgehen, aber mit `checkTicket`. Hat das Segment kein Präfix, greift der Legacy-Pfad: Die ID wird unverändert als Linienticket behandelt. Das stellt sicher, dass ältere QR-Codes ohne Präfix weiterhin funktionieren.

Schlägt der Scan selbst fehl — etwa weil der Nutzer abbricht oder keine Kameraberechtigung erteilt hat — landet man im `catch`-Block. `boxColor` wird auf `'red'` gesetzt.

Listing 17: DetailComponent: QR-Scan mit Präfix-Erkennung

```
1  async startScan() {
2    this.successInfo = undefined;
3    try {
4      const result = await CapacitorBarcodeScanner.scanBarcode({
5        hint: CapacitorBarcodeScannerTypeHint.ALL,
6        cameraDirection: CapacitorBarcodeScannerCameraDirection.BACK
7      });
8
9      const fullUrl = result.ScanResult;
10     const parts = fullUrl.split('/');
11     const lastSegment = parts.pop() || parts[parts.length - 1];
12
13     if (!lastSegment) {
14       this.error = 'Invalid QR format';
15       this.boxColor = 'red';
16       return;
17     }
18
19     if (lastSegment.startsWith('E')) {
20       this.isEventTicket = true;
21       this.scannedValue = lastSegment.substring(1);
22       this.boxColor = 'white';
23       this.checkEventTicket(this.scannedValue);
24     } else if (lastSegment.startsWith('L')) {
25       this.isEventTicket = false;
26       this.scannedValue = lastSegment.substring(1);
27       this.boxColor = 'white';
28       this.checkTicket(this.scannedValue);
29     } else {
30       this.isEventTicket = false;
31       this.scannedValue = lastSegment;
32       this.boxColor = 'white';
33       this.checkTicket(this.scannedValue);
34     }
35   } catch (err) {
36     this.error = 'Scan failed';
37     this.boxColor = 'red';
38   }
39 }
```

confirm Die Methode `confirm` ist ein State-Router. Sie hat keine eigene Geschäftslogik, sondern leitet je nach Zustand weiter.

Ist `firstScanDone` gleich `false`, wurde noch kein gültiges Ticket gescannt. Dann ruft `confirm` die Methode `startScan` auf. Kommt ein gültiges Ticket zurück, setzt `startScan` intern `firstScanDone` auf `true`.

Ist `firstScanDone` gleich `true`, liegt ein geprüftes Ticket vor. Jetzt entscheidet `isEventTicket`: Bei Veranstaltungstickets wird `sendEventValidationToServer` aufgerufen, bei Linientickets `sendValidationToServer`. Beide schicken die Ticket-ID und die eingestellte Personenanzahl ans Backend. Bei Erfolg setzt die App alles zurück — `firstScanDone` auf `false`, Counter auf 0, Ticket-Objekt gelöscht, `boxColor` auf `'blue'`. Danach ist sie bereit für den nächsten Scan.

Der Zweischritt-Mechanismus verhindert, dass ein Kontrolleur versehentlich entwertet, ohne die Ticketdaten geprüft zu haben. Der Button sitzt in beiden Phasen an derselben Stelle, macht aber etwas anderes.

Listing 18: DetailComponent: State-Router für Scan und Entwertung

```
1  async confirm() {
2    if (!this.firstScanDone) {
3      await this.startScan();
4    } else {
5      if (this.isEventTicket) {
6        this.sendEventValidationToServer();
7      } else {
8        this.sendValidationToServer();
9      }
10   }
11 }
```

checkTicket Die Methode `checkTicket` nimmt eine Ticket-ID und fragt das Ticket über den `TicketService` ab. Der schickt einen GET-Request an `/api/tickets/check/{id}` und gibt ein `Observable<LineTicket>` zurück.

Kommt eine Antwort, folgt die Linien-Prüfung: Die `lineId` aus dem Ticket wird mit der `itemId` der aktuellen Session verglichen. Stimmen sie nicht überein, gehört das Ticket zu einer anderen Linie. Die Methode verwirft es, setzt `firstScanDone` auf `false` und zeigt eine Warnung mit dem Namen der richtigen Linie. `boxColor` wird `'yellow'` — nicht rot, weil das Ticket existiert, nur hier falsch ist.

Passt die Linie, werden die Ticketdaten gespeichert, der Counter auf 1 gesetzt, `firstScanDone` auf `true` und `boxColor` auf `'green'`. Die App zeigt die Ticketdetails an und wartet auf Bestätigung.

Im Fehlerfall unterscheidet die Methode zwei Fälle. Bei HTTP 404 erscheint „Ticket existiert nicht" — entweder gelöscht oder falsche ID. Bei allem anderen kommt eine generische Server-Fehlermeldung. In beiden Fällen: `firstScanDone` zurück auf `false`, `boxColor` auf `'red'`.

Listing 19: DetailComponent: Ticketprüfung mit Linien-Validierung

```
1  checkTicket(ticketId: string) {
2    this.ticketService.checkTicket(ticketId).subscribe({
3      next: (ticket) => {
4        if (this.itemId && String(ticket.lineId) !== String(this.itemId)) {
5          this.ticket = undefined;
6          this.firstScanDone = false;
7          this.error = `Wrong Line! Ticket is for: ${ticket.lineName}`;
8          this.boxColor = 'yellow';
9          return;
10       }
11
12       this.ticket = ticket;
13       this.error = undefined;
14       this.boxColor = 'green';
15       this.counter = 1;
16       this.firstScanDone = true;
17     },
18     error: (err) => {
19       this.ticket = undefined;
20       this.firstScanDone = false;
21       this.error = err.status === 404
22         ? 'Ticket does not exist'
23         : 'Server error';
24       this.boxColor = 'red';
25     }
26   });
27 }
```

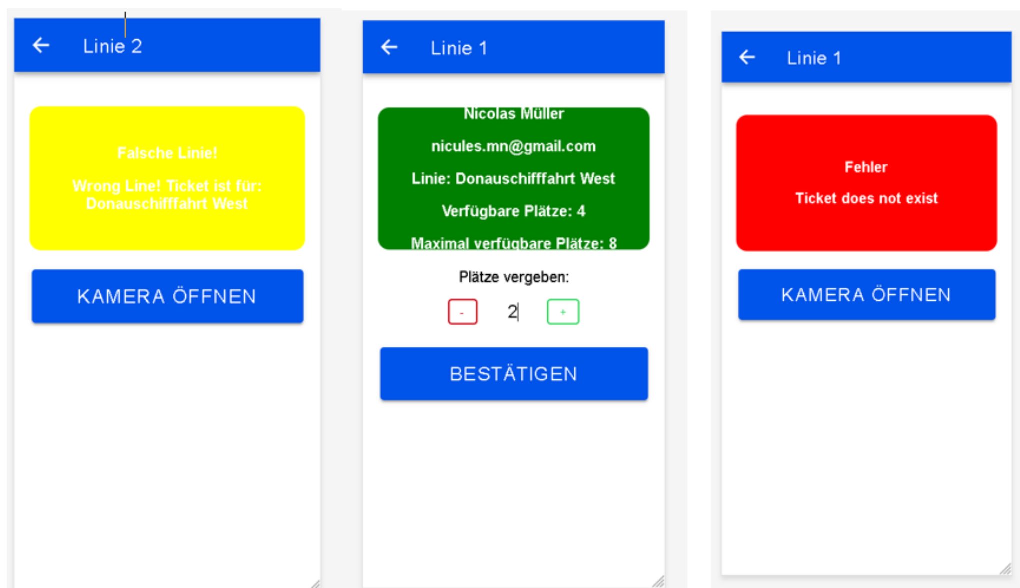

Abbildung 14: Scan-Zustände: gültiges Ticket (grün), falsche Linie (gelb), Fehler (rot)

TicketService

Der `TicketService` wickelt die HTTP-Kommunikation mit den Ticket-Endpunkten ab. Er hat vier Methoden, jeweils zwei für Linien- und Veranstaltungstickets.

`checkTicket` und `checkEventTicket` schicken GET-Requests an `/api/tickets/check/{id}` bzw. `/api/event-tickets/check/{id}`. Zurück kommen typisierte Observables — `Observable<LineTicke` und `Observable<EventTicket>` — sodass die Komponente direkt mit den Modellobjekten arbeitet.

`validateTicket` und `validateEventTicket` schicken PUT-Requests an `/api/tickets/validate/{id}` bzw. `/api/event-tickets/validate/{id}`. Die Personenanzahl geht als Query-Parameter `amount` mit. Der Body ist leer, die Entwertungslogik liegt komplett im Backend.

Eigene Fehlerbehandlung gibt es nicht. Fehler gehen direkt an die Komponente weiter, die sie im `error`-Callback von `subscribe` verarbeitet.

Listing 20: TicketService: API-Wrapper für Ticket-Endpunkte

```
1  @Injectable({ providedIn: 'root' })
2  export class TicketService {
3    private baseUrl = '<server>/api/tickets';
4    private eventBaseUrl = '<server>/api/event-tickets';
5
6    constructor(private http: HttpClient) {}
7
8    checkTicket(ticketId: string): Observable<LineTicket> {
9      return this.http.get<LineTicket>(
10       `${this.baseUrl}/check/${ticketId}`
11     );
12   }
13
14   validateTicket(ticketId: string, amount: number): Observable<any> {
15     return this.http.put(
16       `${this.baseUrl}/validate/${ticketId}?amount=${amount}`, {}
17     );
18   }
19
20   checkEventTicket(ticketId: string): Observable<EventTicket> {
21     return this.http.get<EventTicket>(
22       `${this.eventBaseUrl}/check/${ticketId}`
23     );
24   }
25
26   validateEventTicket(ticketId: string, amount: number): Observable<any> {
27     return this.http.put(
28       `${this.eventBaseUrl}/validate/${ticketId}?amount=${amount}`, {}
29     );
30   }
31 }
```

6.3.4 BackOffice

Das Backoffice ist eine Verwaltungsoberfläche für Administratoren der ÖGEG. Damit kann man Linien und Veranstaltungen verwalten. Man hat damit volle Kontrolle über Linien Fahrten, Event Fahrten, Tickets, Rabatte, Reservierungen und Haltestellen. Außerdem kann man sehen, wie viele Tickets verkauft wurden, wie viel Geld man damit verdient hat und wer die Passagiere waren. Mit der Sitzplatzverwaltung kann man Tische anlegen, Tickets zuweisen und Reservierungsanfragen bestätigen oder ablehnen.

Das Backoffice basiert technisch auf dem Jakarta EE Stack. Für die Web-Schicht werden HTTP-Servlets verwendet, die nach dem MVC-Muster aufgebaut sind. Die Servlets kontrollieren Formulardaten und schicken die aufbereiteten Daten an JSP-Views weiter. Die Datenbankanbindung erfolgt über JPA mit dem EntityManager. Dabei werden EntityGraphs eingesetzt, um Lazy-Loading-Probleme zu vermeiden. Die Anmeldung erfolgt über eine Sitzung mit JWT. Der AuthFilter schützt alle Routen unter /admin/ und leitet nicht angemeldete Anfragen zur Login-Seite weiter.

Authentifizierung

AuthFilter.doFilter Die `doFilter`-Methode ist das Sicherheitstor des gesamten Adminbereichs und wird bei jedem eingehenden Request unter `/admin/*` automatisch vom Servlet-Container aufgerufen, bevor der eigentliche Servlet Code ausgeführt wird. Sie prüft zunächst, ob der aufgerufene Pfad die Login-Seite selbst ist, in diesem Fall wird die Anfrage ohne weitere Prüfung durchgelassen, damit der Anmeldevorgang nicht in einer Endlosschleife endet. Für alle anderen Pfade wird geprüft, ob eine aktive HTTP-Session existiert und ob darin ein JWT-Token hinterlegt ist. Fehlt die Session oder das Token, wird der Benutzer sofort per Redirect zur Login-Seite weitergeleitet. Nur authentifizierte Benutzer gelangen so überhaupt bis zum Servlet.

Listing 21: AuthFilter: Zugriffsschutz für den Adminbereich

```
1  @Override
2  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
3          throws IOException, ServletException {
4
5      HttpServletRequest  req  = (HttpServletRequest)  request;
6      HttpServletResponse resp = (HttpServletResponse) response;
7
8      String uri = req.getRequestURI();
9      boolean isLoginPage = uri.endsWith("/admin/login");
10
11     if (isLoginPage) {
12         chain.doFilter(request, response);
13         return;
14     }
15
16     var session = req.getSession(false);
17     boolean loggedIn = session != null && session.getAttribute("jwt") != null;
18
19     if (!loggedIn) {
```

```
20            resp.sendRedirect(req.getContextPath() + "/admin/login");
21            return;
22        }
23
24        chain.doFilter(request, response);
25    }
```

LoginServlet.doPost Die `doPost`-Methode ist die einzige Möglichkeit, eine gültige Admin-Session zu erstellen. Sie liest E-Mail-Adresse und Passwort aus dem Formular und übergibt sie an den `UsersDAO`, der die Zugangsdaten gegen die Datenbank prüft. Schlägt die Verifizierung fehl, wird das Login-Formular erneut angezeigt und eine Fehlermeldung als Request-Attribut gesetzt — ohne dass eine Session angelegt wird. Bei erfolgreicher Anmeldung wird über den `JWTGen` ein Token generiert und in einer neu erstellten HTTP-Session gespeichert. Zusätzlich wird die E-Mail-Adresse in der Session abgelegt, damit sie in der Oberfläche angezeigt werden kann. Anschließend erfolgt ein Redirect zur Linienübersicht als Startseite des Adminbereichs.

Listing 22: LoginServlet: Verarbeitung des Login-Formulars

```
1   @Override
2   protected void doPost(HttpServletRequest req, HttpServletResponse resp)
3           throws ServletException, IOException {
4       String email    = req.getParameter("email");
5       String password = req.getParameter("password");
6
7       Users user = usersDAO.verifyUserCredentials(email, password);
8
9       if (user == null) {
10          req.setAttribute("error", "Invalid email or password.");
11          req.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(req, resp);
12          return;
13      }
14
15      try {
16          String token = jwtGen.generateToken(user.getId());
17          req.getSession(true).setAttribute("jwt", token);
18          req.getSession().setAttribute("userEmail", user.getEmail());
19          resp.sendRedirect(req.getContextPath() + "/admin/lines");
20      } catch (Exception e) {
21          req.setAttribute("error", "Login failed. Please try again.");
22          req.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(req, resp);
23      }
24  }
```

Linienverwaltung

LinesServlet.doPost Die `doPost`-Methode behandelt zwei Aktionen über denselben Endpunkt und entscheidet anhand des `_action`-Parameters, welche Logik ausgeführt wird. Bei `add-line` wird ein `LineRequest`-Objekt mit Name, Abfahrtszeit und optionalem Preis befüllt und über den `LineDAO` persistiert. Bei `add-stop` wird einer bestehenden Linie eine neue Haltestelle hinzugefügt, indem Port-ID und Abfahrtszeit aus dem Formular gelesen und als `StopRequest` an den DAO weitergegeben werden. Beide Aktionen schreiben bei Erfolg eine Flash-Nachricht in die Session, Fehler werden ebenfalls als Flash-Fehlermeldung festgehalten. Anschließend wird per

Redirect zur Linienübersicht navigiert, um doppelte Formularübermittlungen beim Neuladen zu verhindern.

Listing 23: LinesServlet: Linie und Haltestelle hinzufügen

```
1  @Override
2  protected void doPost(HttpServletRequest req, HttpServletResponse resp)
3          throws ServletException, IOException {
4
5      String action = req.getParameter("_action");
6
7      if ("add-line".equals(action)) {
8          LineRequest lineReq = new LineRequest();
9          lineReq.setName(req.getParameter("name"));
10         lineReq.setDepartureTime(req.getParameter("departureTime"));
11
12         String priceParam = req.getParameter("price");
13         if (priceParam != null && !priceParam.isBlank()) {
14             try { lineReq.setPrice(Double.parseDouble(priceParam)); } catch
                   (NumberFormatException ignored) {}
15         }
16
17         try {
18             lineDAO.addLine(lineReq);
19             req.getSession().setAttribute("flash", "Line created successfully.");
20         } catch (Exception e) {
21             req.getSession().setAttribute("flashError", "Failed to create line: " +
                   e.getMessage());
22         }
23
24     } else if ("add-stop".equals(action)) {
25         long lineId = Long.parseLong(req.getParameter("lineId"));
26
27         StopRequest stopRequest = new StopRequest();
28         stopRequest.setPort(Long.parseLong(req.getParameter("portId")));
29
30         String timeParam = req.getParameter("time");
31         if (timeParam != null && !timeParam.isBlank()) {
32             stopRequest.setTime(java.time.LocalTime.parse(timeParam));
33         }
34
35         try {
36             lineDAO.addStop(lineId, stopRequest);
37             req.getSession().setAttribute("flash", "Stop added.");
38         } catch (Exception e) {
39             req.getSession().setAttribute("flashError", "Failed to add stop: " +
                   e.getMessage());
40         }
41     }
42     resp.sendRedirect(req.getContextPath() + "/admin/lines");
43 }
```

LineDetailServlet.doGet Die `doGet`-Methode bereitet die vollständige Detailansicht einer einzelnen Linie vor und bringt dabei Daten aus mehreren Quellen zusammen. Zuerst wird die Linien-ID aus der URL gelesen und die Linie samt ihren Haltestellen aus der Datenbank geladen. Existiert die Linie nicht, wird eine Fehlermeldung gesetzt und zur Übersicht weitergeleitet. Anschließend werden alle Tickets der Linie abgerufen und daraus Statistiken berechnet: Gesamtanzahl der verkauften Tickets, die Differenz zwischen Soll- und Ist-Menge sowie der Gesamtumsatz. Für jede Haltestelle wird eine Liste aufgebaut, in der steht, wer ein- und aussteigt. Hin- und Rückfahrt-Tickets werden dabei doppelt berücksichtigt, da der Passagier an beiden Haltestellen sowohl als Ein- als auch als Aussteigender gilt. Alle aufbereiteten Daten werden als Request-Attribute gesetzt und an die `line-detail.jsp` weitergegeben.

Listing 24: LineDetailServlet: Detailansicht einer Linie

```java
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    long lineId = parseLineId(req, resp);
    if (lineId < 0) return;

    Line line = lineDAO.findLine(lineId);
    if (line == null) {
        req.getSession().setAttribute("flashError", "Line not found.");
        resp.sendRedirect(req.getContextPath() + "/admin/lines");
        return;
    }

    List<LineTicket> tickets = lineDAO.getTicketsForLine(lineId);

    int totalTickets = 0;
    int totalDeducted = 0;
    double totalRevenue = 0;
    for (LineTicket t : tickets) {
        totalTickets += t.getIs_amount();
        totalDeducted += t.getIs_amount() - t.getShould_amount();
        totalRevenue += t.getPrice() * t.getIs_amount();
    }

    Map<Long, StopPassengersResponse> passengersByStop = new HashMap<>();
    for (Stops stop : line.getStops()) {
        List<LineTicket> stopTickets = lineTicketDAO.getPassengersForStop(stop.getId());
        StopPassengersResponse spr = new StopPassengersResponse();
        for (LineTicket t : stopTickets) {
            boolean isEntry = t.getEntryStop() != null && t.getEntryStop().getId() ==
                stop.getId();
            boolean isExit  = t.getExitStop()  != null && t.getExitStop().getId()  ==
                stop.getId();
            StopPassengersResponse.PassengerEntry entry = new
                StopPassengersResponse.PassengerEntry();
            entry.firstname = t.getFirstname();
            entry.lastname  = t.getLastname();
            entry.amount    = t.getShould_amount();
            entry.roundTrip = t.isRoundTrip();
            if (isEntry) spr.boarding.add(entry);
            if (isExit)  spr.disembarking.add(entry);
            if (t.isRoundTrip() && isExit)  spr.boarding.add(entry);
            if (t.isRoundTrip() && isEntry) spr.disembarking.add(entry);
        }
        passengersByStop.put(stop.getId(), spr);
    }

    req.setAttribute("line", line);
    req.setAttribute("tickets", tickets);
    req.setAttribute("totalTickets", totalTickets);
    req.setAttribute("totalDeducted", totalDeducted);
    req.setAttribute("totalRevenue", totalRevenue);
    req.setAttribute("passengersByStop", passengersByStop);
    req.getRequestDispatcher("/WEB-INF/views/line-detail.jsp").forward(req, resp);
}
```

LineDetailServlet.doPost Die `doPost`-Methode verwaltet die destruktiven Operationen auf der Liniendetailseite und reagiert auf zwei Aktionen. Bei `delete-line` wird zunächst geprüft, ob noch Tickets mit der Linie verknüpft sind — ist das der Fall, wird die Löschung verweigert, da aktive Tickets auf die Linie verweisen und das Löschen die Datenkonsistenz gefährden würde. Sind keine Tickets vorhanden, wird die Linie entfernt und zur Linienübersicht weitergeleitet. Bei `delete-stop` wird eine Haltestelle anhand ihrer ID gelöscht, ohne weitere Vorabprüfung. Alle Aktionen sind in einem Try-Catch-Block abgesichert, sodass Fehler als Flash-Fehlermeldung angezeigt werden.

Listing 25: LineDetailServlet: Linie und Haltestelle löschen

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    long lineId = parseLineId(req, resp);
    if (lineId < 0) return;

    String action = req.getParameter("_action");

    try {
        if ("delete-line".equals(action)) {
            if (lineDAO.getTicketCount(lineId) > 0) {
                req.getSession().setAttribute("flashError", "Cannot delete line
                    tickets still exist.");
            } else {
                lineDAO.deleteLine(lineId);
                req.getSession().setAttribute("flash", "Line deleted.");
                resp.sendRedirect(req.getContextPath() + "/admin/lines");
                return;
            }
        } else if ("delete-stop".equals(action)) {
            long stopId = Long.parseLong(req.getParameter("stopId"));
            lineDAO.deleteStop(stopId);
            req.getSession().setAttribute("flash", "Stop removed.");
        }
    } catch (Exception e) {
        req.getSession().setAttribute("flashError", "Error: " + e.getMessage());
    }

    resp.sendRedirect(req.getContextPath() + "/admin/lines/" + lineId);
}
```

Eventverwaltung

EventsServlet.doPost Die `doPost`-Methode des `EventsServlet` erstellt eine neue Veranstaltung anhand der übermittelten Formulardaten und speichert sie über den `EventDAO` in der Datenbank. Sie liest alle Felder einzeln aus dem Request — darunter Name, Startzeitpunkt als `LocalDateTime`, den zugehörigen Startport sowie Normal- und Studentenpreis als `double`. Jeder Parameter wird separat geprüft: Ist der Wert leer oder ungültig, wird er stillschweigend ignoriert, damit ein einzelnes Fehleingabefeld nicht den gesamten Vorgang abbricht. Bei Erfolg wird eine Bestätigungsmeldung in der Session gesetzt, bei einem Fehler eine entsprechende Fehlermeldung. Abschließend wird per Redirect zur Veranstaltungsübersicht navigiert.

Listing 26: EventsServlet: Neue Veranstaltung erstellen

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    String action = req.getParameter("_action");

    if ("add-event".equals(action)) {
        EventRequest eventReq = new EventRequest();
        eventReq.name = req.getParameter("name");

        String dateTimeParam = req.getParameter("dateTime");
        if (dateTimeParam != null && !dateTimeParam.isBlank()) {
            try {
                eventReq.startTime = LocalDateTime.parse(dateTimeParam);
            } catch (Exception ignored) {}
        }

        String portIdParam = req.getParameter("portId");
```

```
19          if (portIdParam != null && !portIdParam.isBlank()) {
20              try {
21                  eventReq.startPort = Long.parseLong(portIdParam);
22              } catch (NumberFormatException ignored) {}
23          }
24
25          String priceNormalParam = req.getParameter("priceNormal");
26          if (priceNormalParam != null && !priceNormalParam.isBlank()) {
27              try {
28                  eventReq.priceNormal = Double.parseDouble(priceNormalParam);
29              } catch (NumberFormatException ignored) {}
30          }
31
32          String priceStudentsParam = req.getParameter("priceStudents");
33          if (priceStudentsParam != null && !priceStudentsParam.isBlank()) {
34              try {
35                  eventReq.priceStudents = Double.parseDouble(priceStudentsParam);
36              } catch (NumberFormatException ignored) {}
37          }
38
39          try {
40              eventDAO.addEvent(eventReq);
41              req.getSession().setAttribute("flash", "Veranstaltung erfolgreich erstellt.");
42          } catch (Exception e) {
43              req.getSession().setAttribute("flashError", "Fehler beim Erstellen: " +
                    e.getMessage());
44          }
45      }
46
47      resp.sendRedirect(req.getContextPath() + "/admin/events");
48  }
```

EventDetailServlet.doGet Die `doGet`-Methode bereitet die Detailansicht einer Veranstaltung vor. Sie liest die Event-ID aus der URL und lädt die Veranstaltung aus der Datenbank. Wenn die Veranstaltung nicht existiert, erscheint eine Fehlermeldung und man wird zur Übersicht weitergeleitet.

Dann werden alle wichtigen Daten gesammelt. Hier sind Tickets für die Veranstaltung, alle Tische und Reservierungen. Die Reservierungen werden intern in einer Map gespeichert. Dort sind die Reservierungen für jeden Tisch einzeln gespeichert.

Aus den Tickets werden Statistiken berechnet. Die Soll-Menge, die Ist-Menge und der Gesamtumsatz. Alle aufbereiteten Daten werden als Request-Attribute an die JSP-View weitergegeben.

Listing 27: EventDetailServlet: Detailansicht einer Veranstaltung

```
1   @Override
2   protected void doGet(HttpServletRequest req, HttpServletResponse resp)
3           throws ServletException, IOException {
4
5       long eventId = parseEventId(req, resp);
6       if (eventId < 0) return;
7
8       Event event = eventDAO.findEvent(eventId);
9       if (event == null) {
10          req.getSession().setAttribute("flashError", "Veranstaltung nicht gefunden.");
11          resp.sendRedirect(req.getContextPath() + "/admin/events");
12          return;
13      }
14
15      List<EventTicket> tickets = eventTicketDAO.getTicketsPerEvent(eventId);
16      List<Seat> seats = seatDAO.getSeatsForEvent(eventId);
17      List<Reservation> reservations = reservationDAO.getReservationsByEvent(eventId);
18
```

```
19        Map<Long, List<Reservation>> reservationsBySeat = new LinkedHashMap<>();
20        for (Reservation r : reservations) {
21            if (r.getSeat() != null) {
22                reservationsBySeat.computeIfAbsent(r.getSeat().getId(), k -> new
                      ArrayList<>()).add(r);
23            }
24        }
25
26        int totalTickets = 0;
27        int totalDeducted = 0;
28        double totalRevenue = 0;
29        for (EventTicket t : tickets) {
30            totalTickets += t.getShould_amount();
31            totalDeducted += t.getIs_amount();
32            totalRevenue += t.getPrice() * t.getShould_amount();
33        }
34
35        req.setAttribute("event", event);
36        req.setAttribute("tickets", tickets);
37        req.setAttribute("seats", seats);
38        req.setAttribute("reservationsBySeat", reservationsBySeat);
39        req.setAttribute("totalTickets", totalTickets);
40        req.setAttribute("totalDeducted", totalDeducted);
41        req.setAttribute("totalRevenue", totalRevenue);
42        req.getRequestDispatcher("/WEB-INF/views/event-detail.jsp").forward(req, resp);
43    }
```

EventDetailServlet.doPost Die `doPost`-Methode des `EventDetailServlet` ist der zentrale Einstiegspunkt für alle administrativen Änderungen an einer Veranstaltung. Sie liest den Parameter `_action` aus dem Formular und verzweigt je nach Wert in die entsprechende Logik.

Folgende Aktionen werden unterstützt:

- **delete-event** – Löscht eine Veranstaltung, sofern keine Tickets mehr vorhanden sind.
- **add-seat** – Legt einen neuen Tisch mit Nummer und Kapazität an.
- **delete-seat** – Entfernt einen Tisch, sofern er keinem Ticket zugewiesen ist.
- **assign-seat** – Weist einem Ticket einen bestimmten Tisch zu.
- **remove-seat** – Entfernt die Tischzuweisung eines Tickets.
- **confirm-reservation** – Bestätigt eine offene Reservierungsanfrage.
- **reject-reservation** – Lehnt eine Reservierungsanfrage ab.

Löschoperationen sind dabei abgesichert: Eine Veranstaltung kann nur entfernt werden, wenn keine Tickets mehr existieren, ein Tisch nur dann, wenn er keinem Ticket zugewiesen ist. Nach jeder Aktion wird eine Flash-Nachricht in der Session gesetzt und der Browser per Redirect zur selben Detailseite weitergeleitet, um doppelte Formularübermittlungen zu verhindern.

Listing 28: Eventdetail-Verwaltung: Aktionen für Daten

```
1   @Override
2       protected void doPost(HttpServletRequest req, HttpServletResponse resp)
3               throws ServletException, IOException {
4
5           long eventId = parseEventId(req, resp);
```

```
6         if (eventId < 0) return;
7
8         String action = req.getParameter("_action");
9
10        try {
11            if ("delete-event".equals(action)) {
12                if (eventTicketDAO.getTicketCount(eventId) > 0) {
13                    req.getSession().setAttribute("flashError", "Veranstaltung kann nicht
                          gelöscht werden    Tickets sind noch vorhanden.");
14                } else {
15                    eventDAO.deleteEvent(eventId);
16                    req.getSession().setAttribute("flash", "Veranstaltung gelöscht.");
17                    resp.sendRedirect(req.getContextPath() + "/admin/events");
18                    return;
19                }
20            } else if ("add-seat".equals(action)) {
21                int seatNumber = Integer.parseInt(req.getParameter("seatNumber"));
22                int capacity = req.getParameter("capacity") != null ?
                      Integer.parseInt(req.getParameter("capacity")) : 4;
23                seatDAO.addSeat(eventId, seatNumber, capacity);
24                req.getSession().setAttribute("flash", "Tisch " + seatNumber + "
                      hinzugefügt.");
25            } else if ("delete-seat".equals(action)) {
26                long seatId = Long.parseLong(req.getParameter("seatId"));
27                if (seatDAO.isSeatAssigned(seatId)) {
28                    req.getSession().setAttribute("flashError", "Tisch kann nicht gelöscht
                          werden    er ist einem Ticket zugewiesen.");
29                } else {
30                    seatDAO.deleteSeat(seatId);
31                    req.getSession().setAttribute("flash", "Tisch gelöscht.");
32                }
33            } else if ("assign-seat".equals(action)) {
34                UUID ticketId = UUID.fromString(req.getParameter("ticketId"));
35                long seatId = Long.parseLong(req.getParameter("seatId"));
36                eventTicketDAO.assignSeat(ticketId, seatId);
37                req.getSession().setAttribute("flash", "Tisch zugewiesen.");
38            } else if ("remove-seat".equals(action)) {
39                UUID ticketId = UUID.fromString(req.getParameter("ticketId"));
40                long seatId = Long.parseLong(req.getParameter("seatId"));
41                eventTicketDAO.removeSeat(ticketId, seatId);
42                req.getSession().setAttribute("flash", "Tischzuweisung entfernt.");
43            } else if ("confirm-reservation".equals(action)) {
44                UUID reservationId = UUID.fromString(req.getParameter("reservationId"));
45                reservationDAO.confirmReservation(reservationId);
46                req.getSession().setAttribute("flash", "Reservierung bestätigt.");
47            } else if ("reject-reservation".equals(action)) {
48                UUID reservationId = UUID.fromString(req.getParameter("reservationId"));
49                reservationDAO.rejectReservation(reservationId);
50                req.getSession().setAttribute("flash", "Reservierung abgelehnt.");
51            }
52        } catch (Exception e) {
53            req.getSession().setAttribute("flashError", "Fehler: " + e.getMessage());
54        }
55
56        resp.sendRedirect(req.getContextPath() + "/admin/events/" + eventId);
57    }
```

EventDetailServlet.parseEventId

7 Ergebnis

Das Ergebnis des Projekts besteht aus folgenden Teilen:

7.1 Backend

Das Backend versorgt alle visuellen Schnittstellen mit den notwendigen Daten und speichert diese auch in der Datenbank. Des weiteren berechnet das Backend auch die einzelnen Preise der Tickets und berücksichtigt dementsprechend auch die Rabatte. Zum Schluss generiert das Backend noch den QR-Code und sendet diesen per Email an den Kunden.

7.2 BackOffice

Das BackOffice ist die Verwaltungsoberfläche für Administratoren. Hier werden Linien mit Haltestellen angelegt, Veranstaltungen erstellt und Tickets eingesehen. Auch Tischzuweisungen und Reservierungen laufen darüber. Umgesetzt ist das Ganze mit Jakarta EE Servlets und JSP-Views, abgesichert durch einen AuthFilter mit JWT-Sessions.

7.3 Online Shop

Der Online Shop stellt die Schnittstelle zum Kunden dar. Kunden können Produkte einsehen, auswählen und bestellen. Dabei sorgt der Shop für eine einfache Navigation, eine übersichtliche Darstellung der Produkte und sichere Abwicklung der Bestellungen. Alle Aktionen im Shop werden über das Backend verarbeitet, um Daten konsistent zu halten.

7.4 Validierungs App

Mit der Validierungs-App scannen Kontrolleure QR-Codes auf Tickets und prüfen deren Gültigkeit. Ein Präfix im QR-Code unterscheidet zwischen Linien- und Eventtickets. Das Ergebnis

wird farbcodiert angezeigt, ungültige oder bereits entwertete Tickets sind sofort erkennbar. Die App läuft auf Ionic/Angular und kommuniziert per HTTP mit dem Backend.

8 Resümee

8.1 Backend

Die Datenbank planen und aufsetzen war unsere erste Aufgabe, die dank dem DBI Unterricht an der HTL Perg kein Problem war. Die API haben wir uns für Java mit Jakarta EE entschieden da wir diese Methode ausführlich im POSE Unterricht behandelt haben. Der QR Code Generator und der Mail Sender wurden mittels Selbststudium programmiert und hat uns dadurch Wissen, welches wir in der Zukunft benötigen können, vermittelt.

8.2 BackOffice

Das BackOffice ist die Verwaltungsoberfläche für Administratoren. Technisch basiert es auf Jakarta EE Servlets mit JSP-Views, einem AuthFilter und JWT-Sessions. Am aufwändigsten war die Konsistenz bei Linien- und Eventverwaltung: Linien dürfen nur gelöscht werden, wenn keine Tickets daran hängen, und Tischzuweisungen oder Reservierungsbestätigungen müssen ohne Datenverlust durchlaufen.

8.3 Online Shop

Der Online Shop stellt die Kundenoberfläche dar. Wichtig war, die Benutzerfreundlichkeit und die Sicherheit der Daten zu gewährleisten. Außerdem mussten die Prozesse mit dem Backend korrekt verknüpft werden, damit Bestellungen zuverlässig verarbeitet werden.

8.4 Validierungs App

Die Validierungs-App läuft auf Ionic/Angular und wird von Kontrolleuren verwendet, um QR-Codes auf Tickets zu scannen. Beim Scan erkennt die App anhand eines Präfixes, ob es sich um ein Linien- oder Eventticket handelt, und zeigt das Ergebnis farbcodiert an. Fehlerfälle wie ungültige oder bereits entwertete Tickets mussten sauber abgefangen werden, damit der Kontrolleur sofort sieht, was passiert ist.

9 Aufgabenverteilung

9.1 Michael Lintner

Als Projektleiter war ich hauptverantwortlich für die Planung und Kommunikation mit unserem Betreuer und Auftragsgeber. Als Entwickler im Backend war ich zuständig für:

- Implementierung des Backends
- Planung der Datenbank
- Migrieren vom Lokalen System auf den Produktivserver

Verfassen der schriftlichen Teile der Diplomarbeit

- Implementierung des Backends
- Planung der Datenbank
- Migrieren vom lokalen System auf den Produktivserver

Verfassen der schriftlichen Teile der Diplomarbeit

- Grundlagen und Methoden
  - Eidesstattliche Erklärung
  - Kapitel 4.1.1 : Java
  - Kapitel 4.2.3 : IntelliJ IDEA
  - Kapitel 4.3.2 : JAX-RS
  - Kapitel 4.3.3 : JSON-B
  - Kapitel 4.3.4 : Hibernate
  - Kapitel 4.3.5 : jose4j
  - Kapitel 4.3.6 : jBCrypt
  - Kapitel 4.3.7 : Jakarta Mail
- Planung und Realisierung

  - Kapitel 5.2 : Projektstrukturplan
  - Kapitel 5.3 : Gantt-Plan
- Implementierung
  - Kapitel 6.1 : Technischer Überblick
  - Kapitel 6.2 : Backend
- Kapitel 7.1 : Backend
- Kapitel 8.1 : Backend
- Kapitel 9.1 : Michael Lintner
- Anhang A : KI-Nutzung

9.2 Nicolas Müller

Meine Aufgaben umfassten alle Bereiche der Anwendungen, die von den Mitgliedern der ÖGEG im täglichen Betrieb genutzt werden. Dazu gehören das Backoffice, über das Tickets, Linien und Haltestellen verwaltet werden, aber auch die Kontrolleur-App, mit der Tickets vor Ort überprüft werden können.

- **Grundlagen und Methoden**
  - Kapitel 4.1.2 : JSP
  - Kapitel 4.1.3 : Ionic
  - Kapitel 4.1.5 : Apache Maven
  - Kapitel 4.1.6 : WildFly
  - Kapitel 4.2.2 : WebStorm
  - Kapitel 4.3.8 : Capacitor Barcode Scanner
  - Kapitel 4.3.9 : JSON Web Tokens
- **Implementierung**
  - Kapitel 6.3.3 : Ticketvalidierungs-App
  - Kapitel 6.3.4 : Backoffice
- Kapitel 7.2 : Backoffice

- Kapitel 7.4 : Validierungs-App
- Kapitel 8.2 : Backoffice
- Kapitel 8.4 : Validierungs-App
- Kapitel 9.2 : Nicolas Müller

9.3 Simon Schatzl

Web-Applikation

- Als Head-Developer der Webapplikation bin ich zuständig für Angelegenheiten im Frontend, sprich Implementierung des Ablaufs innerhalb der Website.
  - Design und Farbschema der Website
  - Buttons und Auswahlmöglichkeiten
  - Redirects der gegebenen Buttons
  - Kommunikation mit dem Backend

Verfassen der schriftlichen Teile der Diplomarbeit

- Kapitel 1 : Danksagungung
- Kapitel 2 : Kurzfassung
- Kapitel 3 : Abstract
- **Grundlagen und Methoden**
  - Kapitel 4.1.4 : React
  - Kapitel 4.1.7 : Docker Desktop
  - Kapitel 4.2.1 : Visual Studio Code
  - Kapitel 4.2.5 : Penpot
  - Kapitel 4.3.1 : Node Package Manager (npm)
- **Planung und Realisierung**
  - Kapitel 5.1 : Projektorganisation
- **Implementierung**

  - Kapitel 6.3.1 : Login- und Registrierungs-Page
  - Kapitel 6.3.2 : Ticketbuchung

- **Ergebnis und Resümee**
  - Kapitel 7.3 : Online Shop
  - Kapitel 8.3 : Resümee: Online Shop

- Kapitel 9.3 : Simon Schatzl

9.4 Thomas Führer

- Thomas Führer hat keinen Beitrag zur Diplomarbeit geleistet, obwohl er Teil der ursprünglichen Projektgruppe war.

Literaturverzeichnis

[1] Dampfschiff Schönbrunn. Abgerufen am 15.01.2026. Online verfügbar: https://www.dokumentationszentrum-eisenbahnforschung.org/dampfschiff-schoenbrunn-oegeg

[2] Dampfschiff Schönbrunn. Abgerufen am 15.01.2026. Online verfügbar: https://de.wikipedia.org/wiki/SchÃűnbrunn_(Schiff)

[3] React Einführung. Abgerufen am 13.12.2025. Online verfügbar: https://entwickler.de/javascript/react-serie-teil-1

[4] React Developer Page. Abgerufen am 09.12.2025. Online verfügbar: https://reactnative.dev

[5] Barcode Scanner. Abgerufen am 16.03.2026. Online verfügbar: https://capacitorjs.com/docs/apis/barcode-scanner

[6] MySQL Introduction. Abgerufen am 22.03.2026. Online verfügbar: https://www.geeksforgeeks.org/mysql/what-is-mysql/

Abbildungsverzeichnis

1 https://www.oegeg.at/termine-2026/termine-schmalspur-steyrtalbahn/ 3
2 https://commons.wikimedia.org/wiki/File:React-icon.svg 8
3 https://de.wikipedia.org/wiki/Visual_Studio_Code 14
4 Projektstrukturplan TicketShip 25
5 Gantt Plan TicketShip 26
6 Aufbau TicketShip 27
7 Schema Datenbank 32
8 Login-Page 37
9 Registrier-Page 38
10 Fahrtenauswahl 44
11 Eingabe der Details 45
12 Eingabe persönlicher Daten 45
13 Zusammenfassung der Eingaben und Preisberechnung 45
14 Scan-Zustände: gültiges Ticket (grün), falsche Linie (gelb), Fehler (rot) 55

Quellcodeverzeichnis

1 Docker Compose Konfiguration für TicketShip . . . 13
2 Auszug aus der package.json Konfiguration . . . 20
3 QR Code Generator . . . 28
4 JWT Generator . . . 28
5 JWT Verifizerer . . . 29
6 Speicherung des JWT-Tokens . . . 41
7 Speicherung des JWT-Tokens . . . 41
8 Registrierung eines neuen Nutzers . . . 42
9 Holen der Event-Liste beim Drücken des Buchungsbuttons . . . 47
10 Sitzplaetze laden nach Event-Auswahl . . . 47
11 Zentrale Funktion zur Hafenwahl . . . 48
12 JSX-Struktur für die Hafenwahl . . . 49
13 Checkbox für die Rückfahrt-Option . . . 49
14 Logik zur Fehleranzeige bei identischen Häfen . . . 50
15 Abfrage der Preise über die API . . . 50
16 Finalisierung der Buchung mit optionaler Reservierung . . . 51
17 DetailComponent: QR-Scan mit Präfix-Erkennung . . . 53
18 DetailComponent: State-Router für Scan und Entwertung . . . 54
19 DetailComponent: Ticketprüfung mit Linien-Validierung . . . 55
20 TicketService: API-Wrapper für Ticket-Endpunkte . . . 56
21 AuthFilter: Zugriffsschutz für den Adminbereich . . . 57
22 LoginServlet: Verarbeitung des Login-Formulars . . . 58
23 LinesServlet: Linie und Haltestelle hinzufügen . . . 59
24 LineDetailServlet: Detailansicht einer Linie . . . 60
25 LineDetailServlet: Linie und Haltestelle löschen . . . 61
26 EventsServlet: Neue Veranstaltung erstellen . . . 61
27 EventDetailServlet: Detailansicht einer Veranstaltung . . . 62
28 Eventdetail-Verwaltung: Aktionen für Daten . . . 63

Anhang

A KI Nutzung

In der nachfolgenden Tabelle wird definiert, wo welche künstliche Intelligenz benutzt wurde

KI basiertes Hilfsmittel	Einsatzform	Betroffene Teile der Arbeit	Bemerkungen
ChatGPT (OpenAI)	Korrektur von Grammatik und Rechtschreibfehler Unterstüzung mit Latex	ganzes Dokument	
Gemini (Google)	Korrektur von Grammatik und Rechtschreibfehler Unterstüzung mit Latex	ganzes Dokument	
Claude (Anthropic)	Korrektur von Grammatik und Rechtschreibfehler Unterstüzung mit Latex	ganzes Dokument	
Claude (Anthropic)	Nicolas nutzte Claude um aus der Excel Liste der Preismatrix SQL Statements zu machen	SQL insert statements	