

Projektarbeit



Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Corona Management System

Fabien Zwick

Matrikelnr: 2082143

Que Linh Phan

Matrikelnr: 2083223

Hochschule für angewandte
Wissenschaften Augsburg

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
[info\(at\)hs-augsburg-de](mailto:info(at)hs-augsburg-de)

Prüfer: Dipl.-Inf. (FH), Dipl.-De Erich Seifert

Abgabedatum: 05.02.2021

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Gliederung der Arbeit	1
2 Planung und Entwurf	2
2.1 Organisation	2
2.2 Aufgabenaufteilung	2
2.3 Anforderungen	3
2.4 Technologien	3
2.4.1 Spring	3
2.4.2 Vaadin	4
2.4.3 Flutter	4
2.4.4 Hibernate	5
2.4.5 PostgreSQL	5
2.4.6 Weitere Bibliotheken	5
2.5 Tools	6
2.5.1 Entwicklungsumgebungen	6
2.5.2 Postman	7
2.6 Entwurf	7
3 Entwicklung	9
3.1 Backend	9
3.1.1 Controller	9
3.1.2 Persistente Daten	10
3.1.3 Services	12
3.1.4 Registrierung und Aktivierung eines Benutzers	13
3.1.5 Authentifizierung und Autorisierung des Mobile-Client-Users	14
3.1.6 Authentifizierung des Owners	16
3.2 Mobile Client	16
3.3 Web UI	18
4 Schritte zur Inbetriebnahme	20
5 Fazit	22
Quellenverzeichnis	23
Literatur	23
Online-Quellen	23

Kapitel 1

Einleitung

1.1 Motivation

Die Idee für dieses Projekt entstand, als wegen Covid-19 Pflicht wurde, bei jedem Restaurantbesuch seine Daten zu hinterlassen. Diese mussten jedes mal aufs Neue auf Papier geschrieben werden. Dass dabei oft derselbe Stift von vielen Gästen benutzt wurde, war auch eher kontraproduktiv. Des Öfteren kam es auch vor, dass Besucher absichtlich falsche Daten angaben. Hierfür muss eine einfache, moderne und kontaktlose Alternative bereitgestellt werden.

Seit der Idee für das Projekt und der tatsächlichen Implementierung verging einige Zeit, sodass diese Idee mittlerweile unter anderem von der Darfichrein GmbH[11] umgesetzt wurde. Obwohl Darfichrein viele nützliche Features anbieten, fallen zwei Dinge eher negativ auf.

Die Nutzung ist zwar für den User am Endgerät kostenlos, für den Betreiber der Örtlichkeit jedoch nicht [8].

Des Weiteren existiert von DarfIchRein keine native Applikation für mobile Endgeräte. Eine reine Webanwendung bietet zwar viele Vorteile, aber auch einige Nachteile. Wenn die Nutzer nicht gerade sehr moderne Smartphones besitzen, kommen sie um die Installation einer zusätzlichen App nicht herum. Um den Code zu scannen ist nämlich eine QR-Code-Scanner App notwendig. Eine native App könnte diesen intern implementieren.

Außerdem fällt die eindeutige Zuordnung eines Benutzers zu einem Endgerät mit nativen Apps leichter. DarfIchRein überprüft nicht, ob die eingegebenen Nutzerdaten valide sind.

Das Ziel dieses Projekts ist es daher, eine kostenlose Open Source Alternative zu erstellen, die Native Apps auf Smartphones verwendet und die die Telefonnummern der Nutzer verwendet, um deren Identität zu bestätigen.

1.2 Gliederung der Arbeit

Die Arbeit beginnt mit einer Beschreibung über den ersten Entwurf und die Planungsmodalitäten des Projekts. In diesem wird beschrieben, wie das Projekt organisiert und aufgeteilt wird, welche Technologien und Tools benutzt werden und was diese genau sind und können. Außerdem werden die Entwürfe und Anforderungen aufgezeigt.

Anschließend folgt der Praktische Teil, in dem die Implementierung anhand von Code-Schnipsel erklärt und mit dem fertigen UI dargestellt wird. Beginnend mit dem Backend werden dort die einzelnen Komponenten beschrieben, um dann den Authentifizierungsfluss zu zeigen. Dann folgt der Frontend Teil mit der Anbindung des Mobile Clients und das Web UI.

Um das Projekt lokal auf einem Rechner starten zu können, werden im nachfolgenden die nötigen Schritte beschrieben.

Zum Schluss folgt ein Fazit, in dem die umgesetzten Anforderungen nochmal zusammengefasst werden mit einem Ausblick, das auf mögliche Verbesserungen und Erweiterungen ergänzt wird.

Kapitel 2

Planung und Entwurf

In diesem Kapitel wird auf die Planung und die ersten Überlegungen bis zum Entwurf des Projekts eingegangen. Zuerst wird die Arbeitsorganisation und Aufgabenverteilung beschrieben. Anschließend wird hinsichtlich der Anforderungen erklärt, welches Framework benutzt und wie das Projekt strukturiert wird.

2.1 Organisation

Die Organisation wurde zum Start des Projekts festgelegt. Dabei wurden Aspekte des agilen Projektmanagements übernommen und eingesetzt. Es gab ein wöchentliches Meeting, das über die Plattform *Discord* umgesetzt wurde, um die Ziele zu besprechen.

Diese Ziele wurden über die Software *Trello* festgehalten. Trello bietet ein Board an, um Use-Cases und Anforderungen in Tasks festzuhalten. Diese konnte man in die Stages *Next*, *In Progress*, *Staged* und *Approved* unterteilen, um somit einen umfassenden Überblick über den Arbeitsprozess zu behalten. Die Anforderungen werden im sogenannten Backlog festgehalten.

Für die Implementierung wurde das Versionsverwaltungssystem *Git* verwendet. Hier wurde nach dem Feature Branch Workflow gearbeitet. Dies bedeutet im Wesentlichen, dass der `develop` und `master` Branch push-protected sind und neue Änderungen daher nur im Zusammenhang mit einem Pull Request angewandt werden können. Für jedes neue Feature wird also ein neuer, eigener Branch erstellt. Sobald dieser fertiggestellt ist, wird ein Merge Request aufgemacht. Nun hat man die Möglichkeit, sich gegenseitig zu reviewen, um die Codequalität konsistent zu halten. Dies hat zudem noch den Vorteil, dass jedes Teammitglied über die neuesten Codeänderungen sofort Bescheid weiß.

2.2 Aufgabenaufteilung

Durch das gemeinsame Schreiben der Projektanforderungen wurden gleich erste Überlegungen zur Softwarearchitektur gemacht. Dabei wurde die Datenbankarchitektur, sowie das Backend Gerüst, also die Datenmodellierung und Rest API, entwickelt. Darüberhinaus wurde die Business Logik aufgeteilt und implementiert.

Beim Frontend war klar, dass dies einfach aufgeteilt werden kann, durch die gleichzeitige Darstellung anhand einer Mobile App und einer Web UI.

2.3 Anforderungen

Die ersten Überlegungen gehen dahingegen mit den Anforderungen ein, die dieses Projekt realisieren soll. Die Grundfunktionalitäten werden hier aufgezeigt und priorisiert.

Anforderung	Gewichtung	Erreicht
Location über Web UI hinzufügen	hoch	✓
QR Code zu Location anzeigen	hoch	✓
Besucher Daten abfragen	hoch	✓
Aktuelle Besucherzahl abfragen	hoch	✓
Filterung der Besucherhistorie anhand des Zeitraums	mittel	X
Registrierung der Firmeneigentümer	mittel	X
Maximale Besucherzahl festlegen	mittel	✓
Push Notifications	niedrig	X
Standard Check-Out Zeit	niedrig	X

Tabelle 2.1: Übersicht der Anforderungen für Firmen / Owner

Anforderung	Gewichtung	Erreicht
Login mit Client	hoch	✓
Registrierung mit Client	hoch	✓
Handynummer Validierung	hoch	✓
Check-In über QR-Code	hoch	✓
Check-Out über Client	hoch	✓
Check-Out automatisch	mittel	X
Login mit GoogleAuth	niedrig	X
Push Notification	niedrig	X

Tabelle 2.2: Übersicht der Anforderungen für Klienten

2.4 Technologien

Für das Projekt wurde Vaadin, ein Open Source Webframework, benutzt. Dazu wurde noch das Framework Spring, speziell Spring Boot und Spring Boot Web, welche vereinfachte Konfigurationen anbieten, noch hinzugefügt. Die Authentifizierung basiert auf Spring Security. Im Weiteren werden die Frameworks genauer beschrieben. Für das Datenbankmanagement wurde PostgreSQL benutzt.

2.4.1 Spring

Das Spring Framework ist ein quelloffenenes Java Framework, das oftmals in der Webentwicklung Anwendung findet. Das Ziel dieses Frameworks ist, die Entwicklung mit Java zu vereinfachen und Programmierpraktiken zu fördern. Dazu stellt es verschiedene Konfigurationen bereit, sodass man als Entwickler den Fokus auf die Applikationen setzen kann. [1]

Zu den wichtigsten Prinzipien gehören hier die

- Dependency Injection: Benötigte Ressourcen und Objekte werden den Objekten zugewiesen. Dies wird entweder mittels einer Annotation oder einem Setter realisiert.

- Aspektorientierte Programmierung: Technische Aspekte wie zum Beispiel Transaktionen und Sicherheit werden isoliert, sodass der Code klar strukturiert wird.

Auf Basis von Spring existiert auch Spring Boot, das nochmal vereinfachte Konfigurationen bereitstellt. Somit ist eine Erweiterung von verschiedenen Dependencies (also Abhängigkeiten) möglich, indem man diese einfach in der `pom.xml` definiert. Spring Boot stellt damit zum Beispiel mittels der Dependency `spring-boot-starter-web` einen Webserver bereit, der per Autokonfiguration dafür sorgt, dass beim Start der Anwendung den Webserver Tomcat konfiguriert und gestartet wird.

2.4.2 Vaadin

Vaadin ist ein Open Source Webframework, das dem Entwickler erlaubt, die gesamte Anwendung in Java zu schreiben. Um das Frontend zu implementieren, basiert Vaadin daher auf Web Components, die man einfach einbinden kann. [14] Somit umfasst das Framework ereignisgesteuerte Programmierung sowie Steuerelemente, womit man daher kein HTML oder JavaScript braucht. Dies hat auch den Vorteil, dass man über die Service Klassen direkt auf die Daten zugreifen kann und der Datenaustausch nicht über eine API erfolgt.

2.4.3 Flutter

Flutter ist ein kostenloses Open Source UI-Framework, das von Google entwickelt wird und im Mai 2017 veröffentlicht wurde. Das Framework ermöglicht das Entwickeln von Applikationen für mehrere Zielplattformen mit nur einer einzigen Codebasis. Die derzeit unterstützen Plattformen sind Android, iOS, Linux, Mac, Google Fuchsia und seit neuestem auch Windows [4]. Ein Herausstellungsmerkmal von Flutter im Vergleich zu anderen Cross-Plattform-Frameworks ist, dass der in der Programmiersprache Dart geschriebene Quellcode *ahead-of-time* (AOT) zu nativen Maschinencode kompiliert wird und somit sehr performant ist. Eine weitere Besonderheit von Flutter ist, dass keine OEM Widgets zur Darstellung benutzt werden, sondern diese mit Googles Rendering Engine Skia[10] selbst erstellt werden. Das Framework benötigt nur einen Canvas und Zugriff auf Input bzw. Sensoren des Geräts. Dadurch sind Entwickler und Designer sehr frei in der Gestaltung ihrer Apps. Es ist z. B. ohne Probleme möglich, Apps im iOS-Stil (Cupertino) auf Android Geräten darzustellen oder auch umgekehrt. Es besteht auch die Möglichkeit komplett eigene Widgets zu entwerfen. Die Hauptkomponenten, aus denen sich Flutter zusammensetzt, sind [2, 3]:

1. Dart Plattform:

Flutter Apps werden in der Programmiersprache Dart geschrieben. Auf Windows, macOS und Linux läuft Flutter in der Dart-Virtual-Machine. Beim Entwickeln und Debuggen sind, durch eine just-in-time Execution-Engine und Hotreload, Änderungen im Quellcode direkt in der App zu sehen. In Releaseversionen der App wird der Quellcode für bessere Performance AOT in nativen Code kompiliert.

2. Flutter-Engine:

Die Flutter-Engine, die hauptsächlich in C++ geschrieben ist, bietet Low-Level-Rendering-Unterstützung unter Verwendung der Skia-Grafikbibliothek von Google an. Zusätzlich gibt es Schnittstellen zu plattformspezifischen SDKs, wie sie von Android und iOS bereitgestellt werden. Die Flutter-Engine implementiert die Kernbibliotheken von Flutter, einschließlich Animationen/Grafik, Datei- und Netzwerk-I/O, Plugin-Architektur und eine Dart-Laufzeit.

3. Foundation-Bibliothek:

Die Foundation-Bibliothek, die ebenfalls in Dart geschrieben ist, stellt grundlegende Klassen und Funktionen bereit, die zum Erstellen von Anwendungen verwendet werden, wie z.

B. APIs zur Kommunikation mit der Engine.

4. Widget-Bibliothek:

Das Framework enthält zwei Sätze an Widgets, die mit bestimmten Designsprachen konform sind. Material Design-Widgets implementieren Googles gleichnamige Designsprache und Cupertino-Widgets implementieren Apples iOS Human-Interface-Richtlinien. Für Windows und den restlichen Plattformen gibt es derzeit noch keine Design-Widgets. Ein Widget ist im Flutter-Progamm die grundlegende Komponente, wobei ein Widget wiederum aus mehreren Widgets bestehen kann. Ein Widget bündelt die Interaktion, Darstellung und Logik innerhalb eines Objekts. Der Aufbau erinnert hierbei an die Softwarebibliothek React.

2.4.4 Hibernate

Hibernate ist ein objektrelationales Abbildungstool für Java[5]. Es stellt ein Framework zur Verfügung, durch das es möglich ist, Objekten mit Attributen und Methoden (Plain Old Java Objects) in relationalen Datenbanken zu speichern und aus Datenbankeinträgen wieder Java Objekte zu erstellen. Die Beziehungen zwischen Java Objekten werden hierbei ebenfalls auf die entsprechende Datenbank-Relation abgebildet. Diese Relationen werden in den Klassen durch Annotationen angeben. Datenbankzugriffe müssen nicht explizit in SQL programmiert werden. Stattdessen werden Datenbank Abfragen durch JDBC[5, 6] in den für eine spezielle Datenbank benötigten SQL-Dialekt übersetzt. Hibernate ermöglicht also unter Angabe des Dialekts die Verwendung verschiedener Datenbankentypen. Zusätzlich zur eigenen nativen API besitzt Hibernate auch eine Implementation der Java-Persistence-API (JPA) Spezifikation. Dadurch kann es in allen Umgebungen benutzt werden, die JPA unterstützten. In diesem Projekt wird Hibernate für die persistente Speicherung von Clients, Owner, Visits, Locations und Tokens in einer PostgreSQL Datenbank verwendet. Der Zugriff auf diese Objekte wird durch vom Spring Framework zur Verfügung gestellten JPA-Repositories durchgeführt.

2.4.5 PostgreSQL

PostgreSQL ist ein Open Source objektrelationales Datenbankmanagementsystem. Postgres ist SQL konform und bietet zudem zahlreiche weitere Features aus, wie zum Beispiel die Foreign Keys zur Verknüpfung zweier Tabellen oder die Möglichkeit komplexe Abfragen zu tätigen. [16] Das System basiert auf dem Client Server Modell und beinhaltet eine zentrale Serverkomponente **postmaster**, die sämtliche Datenbankdateien sowie Verbindungen zum Datenbankserver verwaltet. Um Postgres zu nutzen braucht man lediglich das Softwarepaket **psql**, das ein Client Programm bereitstellt. Somit hat man die Möglichkeit das Datenbankmanagementsystem über die Kommandozeile zu nutzen. Alternativ gibt es zahlreiche grafische Benutzeroberflächen, um eine vereinfachte Darstellung zu erzielen.

2.4.6 Weitere Bibliotheken

Zxing

ZXing (Zebra Crossing) ist eine quelloffene multi-format 1D und 2D Barcode Bildverarbeitungsbibliothek die in Java implementiert wurde[17]. In diesem Projekt wird sie verwendet, um die QR-Codes der verschiedenen Locations zu erstellen.

QR Code Scanner

Dieses Flutter Paket liefert ein Widget und einige Klassen, die das Scannen von QR-Codes auf mobilen Endgeräten ermöglicht[9]. Es ist auf Android und iOS lauffähig, indem die Plattform-

View nativ in Flutter eingebettet wird. Im Projekt wird das Paket verwendet um die QR-Codes der Locations zu scannen.

Twilio

Die Twilio Java SDK ermöglicht die Nutzung des Twilio Webservice. Dieser Service ermöglicht das programmgesteuerte Tätigen und empfangen von Anrufen, senden und empfangen von Nachrichten sowie das ausführen andere Kommunikationsfunktionen. Das SDK wird in diesem Projekt genutzt, um den Client-Usern beim Registrieren den Aktivierungslink auf an ihr Endgerät zu senden.

Lombok

Projekt Lombok[7] ist eine Java-Bibliothek die durch das Angeben von Annotationen dem Entwickler die Erstellung von Boilerplate-Code erspart. Die Klassen bleiben dadurch sehr übersichtlich. Anstatt für jedes Feld der Klasse einen *Getter* und *Setter* zu schreiben, genügt es z.B. die Annotationen `@Getter`, `@Setter` oder `@Data` vor die Klasse zu setzen. Lombok generiert den Code dann automatisch. In diesem Projekt wurde Lombok in fast allen Klassen verwendet.

2.5 Tools

Dieses Projekt wurde mit zwei verschiedenen Entwicklungsumgebungen implementiert. Zum Testen der API eignet sich darüberhinaus eine Software, mittels der man die Endpoints testen kann.

2.5.1 Entwicklungsumgebungen

IntelliJ

IntelliJ ist eine IDE des Softwareunternehmens JetBrains für die Programmiersprachen Java, Kotlin, Groovy und Scala. [12]

Es bietet eine umfangreiche Unterstützung von zum Beispiel Apache Maven und Ant an, die in diesem Projekt benutzt wurden um die Bibliotheken zu installieren und zu bauen. Außerdem bietet die IDE die Möglichkeit an, einfach den Code zu debuggen, das eine angenehme Programmiererfahrung darstellt. Integriert ist auch ein Linter, der Stellen im Code markiert, die nicht den Konventionen entsprechen oder schöner zu implementieren können. Besonders bei Java ist dies angenehm um sofort auf fehlende ; oder andere leichtsinnige Sachen zu verweisen.

VS Code

Visual Studio Code ist ein Open Source Editor von Microsoft. Der Editor basiert auf dem Framework Electron und ermöglicht zahlreiche Funktionen, der mit weiteren Bibliotheken und Plugins einfach erweiterbar ist. Somit ist der Editor für so gut wie jede Programmiersprache kompatibel. [15]

2.5.2 Postman

Postman und Insomnia sind Anwendungen, die eine API Entwicklung erleichtern. Sie beinhalten einen API Client, der REST Requests senden kann und somit testet, ob genau die Daten geschickt und zurückkommt, die man für die weitere Implementationen braucht. [13]

2.6 Entwurf

Der Entwurf der Datenbank sieht vor, dass dieses Projekt aus 4 Tabellen besteht. Der Owner ist der Besitzer der Örtlichkeit und steht daher zu den Locations in einer 1:n Relation. Der Client ist der mobile Enduser, der sich in die Örtlichkeiten ein- und auschecken kann. Um dies zu realisieren gibt es die Besuchertabelle, die die Tabellen Location und Client zusammen joinen. Die Visit Table hat also jeweils die Beziehung 1:n zur Location und zum Client. Damit kann man zu jedem Besuch die Benutzerdaten abfragen, um bei einem Corona Fall die relevanten Daten zu kontaktieren.

Der Datenbankentwurf stimmt mit dem Endergebnis der Datenbankarchitektur überein. Die folgende Abbildung stellt den Entwurf nochmal bildlich dar.

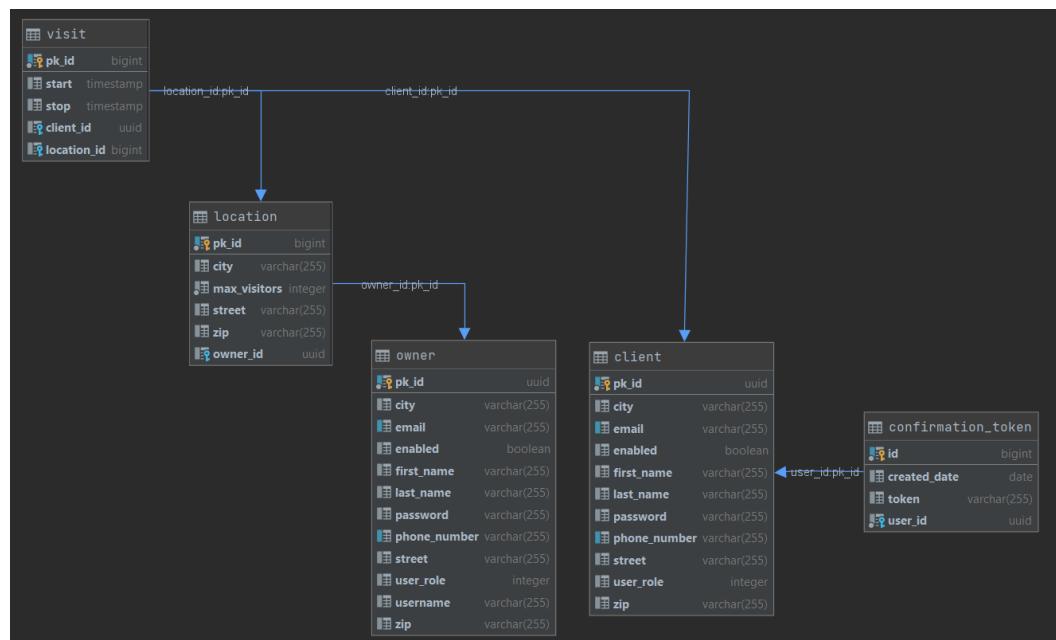


Abbildung 2.1: Datenbankarchitektur

Der Entwurf der UI dient als Vorlage für die bildliche Vorstellung und zum gemeinsamen Verständnis. Dazu wurde händisch skizziert, welche Grundfunktionalitäten wie ausschauen sollen. Nachdem diese von beiden Seiten als genehmigt wurde, fängt auch schon der Implementierungs- teil an.

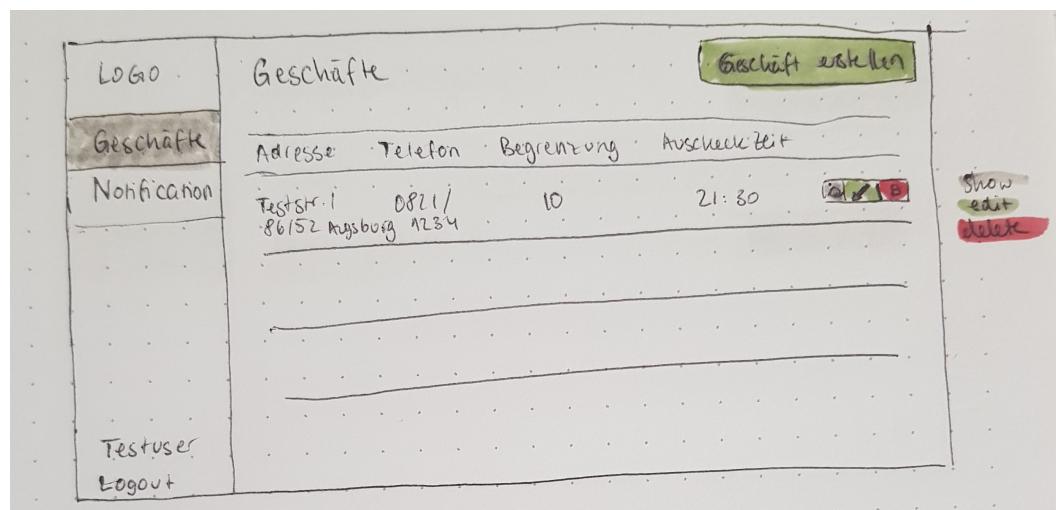


Abbildung 2.2: Entwurf Web UI

Für die Softwarearchitektur ist ein Rest Server geplant, womit man die Mobile Flutter App einfach durch die API Schnittstellen an das Backend anbinden kann. Die Mobile App dient den Endusers, sich für ein Besuch in einer Örtlichkeit anzumelden bzw. ein- und auszuchecken.

Dazu gibt es ein Backend Web Interface, das für die Besitzer der Örtlichkeiten eine Möglichkeit darstellt, Besuche und damit Kontakte nachzuverfolgen, neue Locations zu erstellen und die generierten QR-Codes abzurufen.

Durch diese Anforderungen ist unsere Wahl für die Implementierung der Rest API und Web Frontend auf Vaadin gefallen, um produktiv in Java arbeiten zu können, ohne auf weitere Sprachen wechseln zu müssen wie zum Beispiel JavaScript, da mit der Mobile App bereits schon Dart verwendet wird.

Da der Schwerpunkt dieses Fachs besonderen Wert auf die objektorientierte Softwareentwicklung legt, ist auch dies ein Grund um Vaadin bzw. Spring Boot zu benutzen, um so verschiedene Komponenten aus dem Backend einfach wiederzuverwenden.

Kapitel 3

Entwicklung

In diesem Kapitel wird auf die Entwicklung der verschiedenen Bestandteile des Projekts im Detail eingegangen. Zunächst wird beschrieben wie das Backend, bzw. die Businesslogik implementiert wurde. Danach wird noch auf die Implementierungen des Mobile-Frontends für die Clients und des Web-Frontends für die Owner eingegangen. Hierbei wird das schrittweise Vorgehen, sowie die technischen Probleme gezeigt.

3.1 Backend

3.1.1 Controller

Damit sich ein Client ein- und auschecken, registrieren und anmelden kann, muss das Backend eine Rest-API-Schnittstelle zur Verfügung stellen, mit dem der Mobile-Client kommunizieren kann. Durch das Spring-Boot-Web Framework kann diese Schnittstelle sehr einfach realisiert werden. Es muss lediglich eine Klasse mit der `@RestController` und ihre Methoden mit einer `@RequestMapping` Annotation seiner Wahl versehen werden. Beim `@RequestMapping` kann auch der gewünschte Pfad und die Request-Art (GET, POST, etc.) angegeben werden. Jeder Pfad der Rest-API startet in diesem Projekt mit »`/api/v1`«. Für diese Anwendung wurden die folgenden vier Controller erstellt.

1. ClientController
2. VisitController
3. LocationController
4. OwnerController

Der **ClientController** bietet die Schnittstelle für sämtliche Anfragen, die mit einem User-Client zu tun haben. Im Folgenden ist einen Auflistung der wichtigsten Routen zu sehen:

- `/api/v1/client/sign-up`

Diese Route verwendet der Mobile-Client, um einen Benutzer zu registrieren. Hierbei werden die Nutzerdaten in einem JSON-Body in einem POST-Request gesendet.

- `/api/v1/client/confirm?token=...`

Hat sich ein Benutzer registriert, wird ihm an die angegebene Telefonnummer ein Link mit dieser Route gesendet, der als Request-Parameter einen Token enthält. Durch das Aufrufen dieser Route mit dem angefügten Token wird der Benutzer freigeschaltet. So lange er dies nicht tut, kann er sich nicht anmelden.

- `/api/v1/client/token/{token}`

Wie später in dieser Arbeit beschrieben, erhält ein Benutzer nach erfolgreicher Anmeldung

einen JSON-Web-Token. Dieser Token kann an den angegebenen Pfad als Pfadparameter in einem GET-Request gesendet werden, um Informationen über sich selbst abzufragen.

Der **VisitController** bietet die Schnittstelle für alle Anfragen, die mit dem Ein- und Auschecken zu tun haben. Im Folgenden die Auflistung der vom Mobile-Client verwendeten Routen:

- **/api/v1/visit/checkedin?user=...**

An diesem Endpunkt kann ein Benutzer mit einem GET-Request abfragen, ob er bereits in einer Location eingekickt hat. Die Anfrage muss hierfür einen Request-Parameter mit der User-ID enthalten.

- **/api/v1/visit/checkin?user=...&location=...**

Diese Route wird verwendet, wenn ein Benutzer einchecken möchte. Hierfür benötigt er neben seiner User-ID, auch eine Location-ID. Die erhält er über das Scannen des QR-Codes der Location. Es muss sich um ein POST-Request handeln, der die beiden Request-Paramenter beinhaltet.

- **/api/v1/visit/checkout?user=...**

Durch das Senden einer POST-Request mit der User-ID an diese Route wird dieser, falls er gerade eingekickt ist, aus der Location ausgekickt.

Die **Location-** und **OwnerController** existieren nur aus Debug Gründen. Im Mobile-Client finden diese keine Anwendung. Im Web-UI wird direkt auf die später beschriebenen Services zugegriffen, anstatt auf die Controller.

ControllerAdvisor und Exceptions

Bei Verwendung der API können natürlich auch Fehler auftreten. Zum Beispiel wenn sich ein Benutzer in eine Location einchecken möchte, die nicht existiert. Für solche Fälle wurden einige spezielle RuntimeExceptions implementiert. Außerdem wurde für jeden Controller ein ControllerAdvisor erstellt. Diese Advisor dienen als ExceptionHandler für die im Controller geworfenen Exceptions. Hierfür wurde für jede "Controller"-Exception eine Handler-Methode implementiert. Diese verarbeiten die Exception so, dass dem Mobile-Client eine aussagekräftige Fehlermeldung als Http-Response gesendet wird. In folgender Abbildung ist als Beispiel der Advisor des ClientController mit deren Exceptions zu sehen.

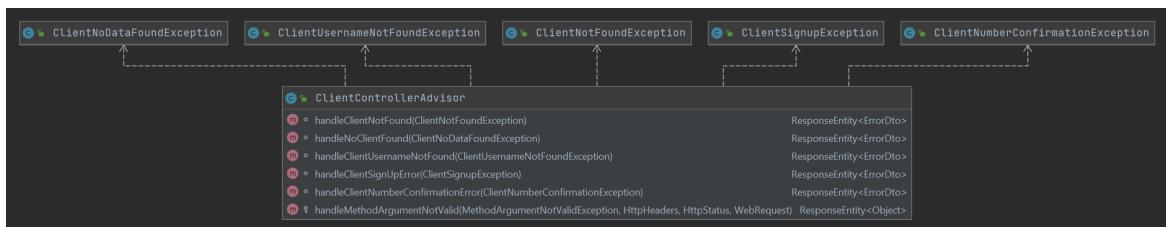


Abbildung 3.1: ClientControllerAdvisor

3.1.2 Persistente Daten

Um Client-, Visit-, Owner-Daten, etc. persistent abzuspeichern, wird eine Datenbank benötigt. In diesem Projekt ist dies eine PostgreSQL Datenbank.

Models

Um Datenbanktabellen und deren Relationen untereinander zu definieren, werden Models verwendet. Instanzen dieser Models repräsentieren Datenbankeinträge als Java-Objekte. Um aus einer Java-Klasse eine Tabelle zu erstellen, muss die Klasse lediglich mit der `@Entity` Annotation gekennzeichnet sein, einen leeren Konstruktor und Getter/Setter für die Felder der Klasse besitzen. Die folgende Abbildung zeigt die in diesem Projekt verwendeten Models.

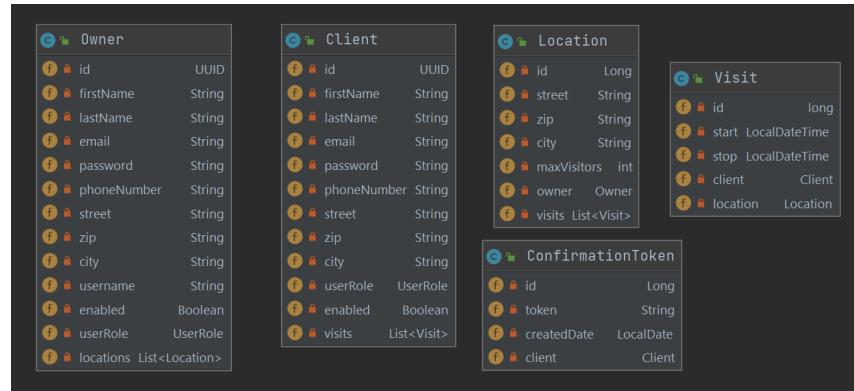


Abbildung 3.2: Models bzw. Entities

Innerhalb dieser Model-Entitäten können Verweise auf andere Entitäten existieren. Wenn dies der Fall ist, muss durch weitere Annotationen der einzelnen Felder festgelegt werden, wie die Beziehung zwischen den Entitäten bzw. wie die Beziehung zwischen den Datenbanktabellen ist. Dadurch wird z. B. bestimmt, in welcher der beiden Tabellen der Fremdschlüssel existieren soll oder ob (bei einer many-to-many Relation) eine neue Tabelle erstellt werden muss. In diesem Projekt gab es die Schwierigkeit, dass sich zwei Entitäten gegenseitig referenziert haben. Beim Versuch diese Entitäten als JSON-Objekt via HTTP zu versenden, kam nach einer langen Wartezeit wegen einer endlosen Rekursion ein sehr großes JSON-Objekt an, das unbrauchbar war. Eine mögliche Lösung für dieses Problem war die Annotation `@JsonManagedReference` über dem problematischen Feld. Eine andere Lösung war die Verwendung der im Folgendem beschriebenen DTOs.

Data-Transfer-Objects

Data-Transfer-Objects (DTO) werden in diesem Projekt verwendet, um Daten zwischen dem Server und dem Mobile-Client auszutauschen. Anstatt hierfür Models zu verwenden, die meist mehr Informationen beinhalten als der Client benötigt, werden mit ihnen DTO Instanzen erstellt und diese anstelle gesendet. Die DTOs beinhalten wirklich nur genau das, was der Mobile-Client benötigt. Unnötige Informationen wie z. B. ob der Benutzer aktiviert wurde oder sein Passwort können hier weggelassen werden. Dies verringert die Netzwerkauslastung. Des Weiteren hat man mehr Kontrolle, als wenn direkt ein Model/Entität gesendet wird. Beim vorherigen Beispiel mit dem Rekursionsproblem kann z.B. explizit angegeben werden, dass anstelle des Objekts nur seine ID übertragen werden soll. In folgendem Quelltextausschnitt wird gezeigt, wie als Antwort auf eine Auscheck-Anfrage ein Visit-DTO anstelle des Models gesendet wird.

```

1 @PostMapping("/checkout")
2 public ResponseEntity<VisitDto> checkOutClient(@RequestParam(name = "user") String userId){
3     Visit v = visitService.checkOutClient(userId);
4     if(v != null)
5         return ResponseEntity.ok(new VisitDto(v));
6     throw new VisitCheckOutException(userId);

```

7 }

Repositories

Um Einträge zur Datenbank hinzuzufügen, zu löschen oder diese zu modifizieren, werden in diesem Projekt die vom Spring-Boot-Data-Framework bereitgestellten Repositories verwendet. Ein Repository für ein Model/Entität kann erstellt werden, in dem man z.B. ein neues Interface erstellt, welches vom `JpaRepository`-Interface erbt. Eine konkrete Implementation des Repository muss nicht erstellt werden, da dies das Framework übernimmt. Das JPA-Repository enthält schon die meistens Standard Methoden, die für das Interagieren mit einer Datenbank benötigt werden. Um eigene SQL-Abfragen verwenden zu können, müssen diese nicht in SQL formuliert werden. Es ist möglich, im eigens erstellten Interface Methoden anzugeben, deren Namen ausgewertet werden und automatisch in SQL-Statements umgewandelt werden. Folgender Codeausschnitt zeigt z. B. wie in der Datenbank ein Visit Eintrag eines Clients abgerufen wird, der noch nicht ausgecheckt hat. Der Rückgabeparameter ist hierbei ein `Optional`, da es sein kann, dass kein solcher Client existiert oder er nicht eingecheckt ist.

```
1 public interface VisitRepository extends JpaRepository<Visit, Long> {
2     Optional<Visit> findVisitByClientAndStopIsNull(Client c);
3 }
```

3.1.3 Services

Die Service-Klassen beinhalten sämtliche Business-Logik. Service-Klassen haben Zugriff auf andere Services und auf die Repositories der Datenbank. Hierbei wurde darauf geachtet, dass keine konkreten Implementierungen von Services verwendet werden, sondern deren Interfaces. Durch die `Dependency Injection` des Frameworks wird für jedes verwendete Interface, das mit `@Autowired` annotiert ist, eine Instanz der Klassen erstellt, die dieses implementiert und mit der `@Primary` Annotation gekennzeichnet ist. Falls bereits eine Instanz existiert, wird diese stattdessen verwendet. Alternativ zur `@Primary` Annotation kann mit der `@Qualifier` Annotation der exakte Name der gewünschten Service-Implementation angegeben werden. Durch dieses Vorgehen kann die konkrete Implementierung eines Service sehr einfach ausgetauscht werden. Dies erhöht die Wartbarkeit des Codes. In folgendem Codeausschnitt ist ein Teil des `VisitService` zu sehen, der die Nutzung der Interfaces verdeutlichen soll. Durch die `@Autowired` Annotation werden die Abhängigkeiten durch das Framework injiziert. Für die Interfaces werden hierbei die korrekten Implementationen instanziert.

```
1 public class VisitService implements IVisitService {
2     private VisitRepository visitRepository;
3     private IClientService clientService;
4     private ILocationService locationService;
5
6     @Autowired
7     VisitService(VisitRepository visitRepository, IClientService clientService,
8                 ILocationService locationService) { ... }
9 }
```

Die folgende Abbildung zeigt alle in diesem Projekt verwendeten Services.

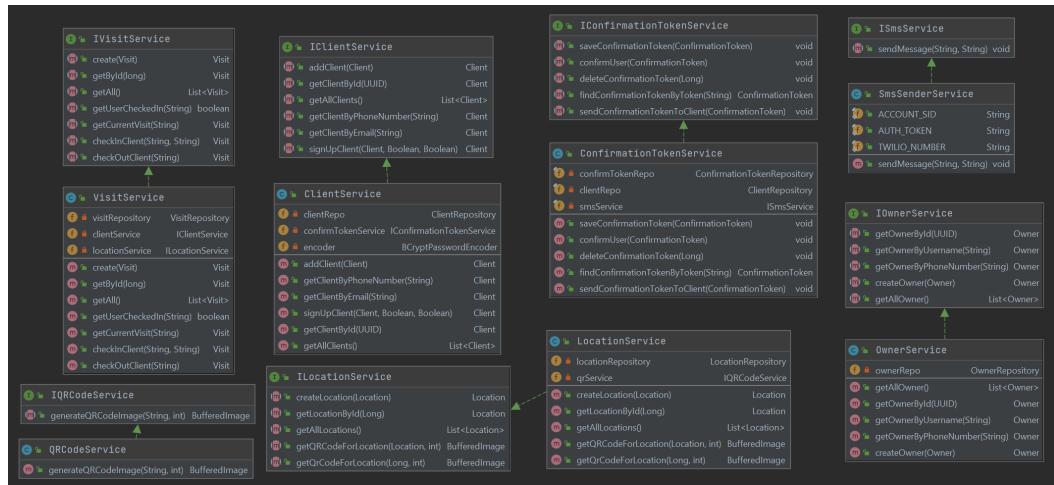


Abbildung 3.3: Services in diesem Projekt

3.1.4 Registrierung und Aktivierung eines Benutzers

Das Registrieren eines neuen Nutzers geschieht in zwei Schritten. Im ersten Schritt ruft der Mobile-Client über die API die `signUp()`-Methode auf. Sind die übermittelten Nutzerdaten vollständige, wird die `signUpClient()`-Methode des `ClientService` aufgerufen. Hierbei kann angegeben werden, ob der User sofort freigeschaltet werden soll und ob ein Bestätigungsstoken gesendet werden soll. Der Service erstellt den neuen Nutzer in der Datenbank und verschlüsselt hierbei sein gewähltes Passwort. Standardmäßig ist hierbei das Enabled-Feld des Nutzers auf `False` gesetzt und solange dies der Fall ist, ist keine Anmeldung möglich. Zusätzlich wird ein `ConfirmationToken` erstellt und in der Datenbank hinterlegt, welcher eindeutig den neuen User zugeordnet wird. Mit der Erstellung des Tokens wird dieser auch an die Telefonnummer des Nutzers gesendet. Hierfür benutzt der `ConfirmationTokenService` wiederum einen `SmsService`, der die Twilio SDK verwendet. Im Folgenden ist der Quellcode der `signUpClient()`-Methode zu sehen, um die einzelnen Schritte noch einmal zu verdeutlichen.

```

1 public Client signUpClient(Client client, Boolean sendToken, Boolean
                           enableIAccountImmediately) {
2     final String encryptedPassword = encoder.encode(client.getPassword());
3     client.setPassword(encryptedPassword);
4     client.setEnabled(enableIAccountImmediately);
5     final Client createdClient = clientRepo.save(client);
6
7     final ConfirmationToken confirmToken = new ConfirmationToken(createdClient);
8     confirmTokenService.saveConfirmationToken(confirmToken);
9
10    if(sendToken == true)
11        confirmTokenService.sendConfirmationTokenToClient(confirmToken);
12    return createdClient;
13}

```

Im zweiten Schritt ruft der Mobile-Client über die API die `confirmPhoneNumber()`-Methode auf und übergibt ihr den Bestätigungsstoken. Existiert dieser Token, wird der ihm zugeordneten User ausfindig gemacht und aktiviert, indem das Enabled-Feld auf `True` gesetzt wird. Von nun an kann sich der Benutzer mit dem Mobile-Client anmelden. Im Folgendem auch der Quellcode zur Veranschaulichung.

```

1 @GetMapping("/confirm")
2 public ResponseEntity confirmPhoneNumber(@RequestParam("token") String token){
3     var t = confirmService.findConfirmationTokenByToken(token);

```

```

4     if(t != null) {
5         confirmService.confirmUser(t);
6         return ResponseEntity.ok(t.getClient().getId());
7     }
8     throw new ClientNumberConfirmationException();
9 }
```

3.1.5 Authentifizierung und Autorisierung des Mobile-Client-Users

Damit nur registrierte Benutzer die API verwenden können, muss es ein Security-System geben, welches für jede Anfrage überprüft, ob der Nutzer berechtigt ist diese auszuführen. In diesem Projekt wird hierfür das Spring-Security-Framework verwendet. Um die Identität eines Benutzers nachzuweisen, werden JSON-Web-Token (JWT) verwendet, da diese zustandslos sind und einfach im Mobile-Client gespeichert werden können. Die Sicherheitseinstellungen werden in der **SecurityConfiguration**-Klasse konfiguriert. Diese besitzt neben globalen Konfigurationen auch noch zwei eingebettete Klassen, die speziell nur für die API und wie später näher beschrieben, nur für die Web-Interface-Security zuständig sind. In der globalen Konfigurationsklasse sind die folgenden zwei Funktionen sehr wichtig.

```

1 @Bean
2 public BCryptPasswordEncoder bCryptPasswordEncoder(){return new BCryptPasswordEncoder();}
3
4 @Autowired
5 protected void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
6     auth.userDetailsService(this.userDetailsService).passwordEncoder(pwEncoder);
7 }
```

Das erste Bean stellt global einen Passwort-Encoder zu Verfügung. Mit der **configureGlobal()**-Methode wird der für die Authentifizierung verwendete **AuthenticationManager** konfiguriert. Dieser erhält eine Implementierung des **UserDetailsService**-Interface und den Encoder, der für die Passwortverschlüsselung verwendet wird. Grob erklärt, benutzt der **AuthenticationManager** bei einem Authentifizierungsversuch den **UserDetailsService**, um Informationen des Users (Username, Passwort, ob Aktiviert, Berechtigungen, usw.) aus der Datenbank abzufragen. Der übergebene Kodierer wird dann benutzt um das Passwort, welches vom Mobile-Client in Klartext übermittelt wird, zu verschlüsseln und mit dem bereits verschlüsselten Passwort in der Datenbank zu vergleichen. Existiert der Benutzer und stimmen beide Passwörter überein, war die Authentifizierung erfolgreich. Genauer wird hierauf beim **JWTAuthenticationFilter**[3.1.5] eingegangen.

Die innere Klasse **ApiWebSecurityConfigurationAdapter** erbt, wie die globale Konfigurationsklasse, ebenfalls von der abstrakten Klasse **WebSecurityConfigurerAdapter**. Diese Konfiguration wird jedoch ausschließlich für die Rest-API verwendet. In der **configure()**-Methode wird das **HttpSecurity**-Objekt so konfiguriert, dass man für einen Zugriff auf die API bis auf wenige Ausnahmen immer eingeloggt sein muss.

Die Ausnahmen bildet hier die Login-Route zum Authentifizieren und die SignUp-Route zum Registrieren des Nutzers. In dieser Konfiguration werden auch die im folgenden beschriebenen zwei essenzielle Filter hinzugefügt.

JWTAuthenticationFilter

Der **JWTAuthenticationFilter** wird immer dann aktiv, wenn ein Mobile-Client versucht, sich über die Login-API-Route anzumelden. Diese Filter-Klasse erbt von der **AbstractAuthenticationProcessingFilter**-Klasse und überschreibt zwei ihrer Methoden. Die **attemptAuthentication()**-Methode wird bei jedem Authentifizierungsversuch aufgerufen. Wie oben bereits kurz beschrieben wird hier der **AuthenticatorManager** verwendet um zu überprüfen, ob der Nutzer authentifiziert werden kann. Wie im folgenden (gekürzten) Codeausschnitt zu

sehen, wird das Ergebnis mittels eines `Authentication`-Objekts zurückgegeben. Alles Weitere, wie das Setzen des `Authentication`-Objekts im `SecurityContext`, wird von der Eltern-Klasse übernommen.

```

1 @Override
2 public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse
   Response){
3     AccountCredentials credentials = //Credentials aus JSON-Body lesen
4     var token = new UsernamePasswordAuthenticationToken(credentials.getUsername(),
   credentials.getPassword());
5     return getAuthenticationManager().authenticate(token);
6 }
```

Die zweite Methode, die überschrieben wird, ist `successfulAuthentication()`. Wie der Name vermuten lässt, wird sie von der Elternklasse aufgerufen, sobald die Authentifizierung erfolgreich war. In dieser wird dem `HttpServletResponse` mithilfe des `TokenAuthenticationService` ein Authentication-Header hinzugefügt, den der Mobile-Client abspeichert und zur weiteren Autorisierung verwendet. Der Service generiert hierfür aus dem Benutzernamen und einem Geheimnis einen JSON-Web-Token.

```

1 @Override
2 protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse
   response, FilterChain chain, Authentication authResult) {
3     String userName = authResult.getName();
4     tokenAuthenticationService.addAuthentication(response, userName);}
```

JWTAuthorizationFilter

Der `JWTAuthorizationFilter` wird bei jeder REST-API-Anfrage aktiv, da nichts mit dem Einloggen und Registrieren zu tun hat. Der Filter überschreibt die `doFilter()`-Methode der `GenericFilterBean`-Elternklasse. In dieser Methode wird mithilfe des `TokenAuthenticationService` überprüft, ob die Anfrage einen Header mit validem JWT besitzt. Der folgende gekürzte Codeausschnitt zeigt, wie der Service überprüft, ob der Benutzer autorisiert werden darf oder nicht.

```

1 public Authentication getAuthentication(HttpServletRequest request)
2 {
3     String token = request.getHeader(SecurityConstants.HEADER_STRING);
4     if (token != null && token.startsWith(SecurityConstants.TOKEN_PREFIX)) {
5         String username = //parse jwt. username is subject
6         if (username != null) {
7             UserDetails u = userDetailsService.loadUserByUsername(username);
8             return new UsernamePasswordAuthenticationToken(u.getUsername(), u.getPassword(),
   u.getAuthorities());}
9     return null;
10 }
```

Genau wie der im vorigen Filter gezeigte `AuthenticationManager` gibt auch diese Methode ein `Authentication`-Objekt zurück. Dieses Objekt wird dem Security-Context übergeben. Wenn das Objekt nicht Null ist, war die Autorisierung erfolgreich und dem Nutzer wird Zugriff auf die für seine Benutzerrolle erlaubten Ressourcen gewährt. Die Vorgehensweise wird mit dem folgenden Codeausschnitt der überschriebenen `doFilter()`-Methode verdeutlicht.

```

1 public void doFilter(ServletRequest Request, ServletResponse Response, FilterChain Chain){
2     Authentication authentication = tokenAuthService.getAuthentication(Request);
3     SecurityContextHolder.getContext().setAuthentication(authentication);
4     filterChain.doFilter(Request, Response);}
```

3.1.6 Authentifizierung des Owners

Wie bereits erwähnt, erfolgt die Authentifizierung des Besitzers bzw Owners über eine eigene interne Klasse in der `SecurityConfiguration`. Die Klasse `WebSecurityConfig` erbt auch hier vom `WebSecurityConfigurerAdapter` und beinhaltet die Einstellung, dass jede Route, die nicht mit `/api/` anfängt, geschützt wird. Eine Ausnahme stellt natürlich die Login Route dar. Ebenfalls nutzt diese Konfiguration die Methode `configureGlobal`, die anhand der Form Daten, die vom Frontend übermittelt werden, abfrägt und mit den Informationen in der Datenbank abgleicht. Die Konfiguration werden anhand der `Order` Annotation unterschieden. Die API Konfiguration hat hier die `@Order(1)`, das heißt, dass diese Methode zuerst angesteuert wird und überprüft, ob die Route mit `/api/` anfängt. Tut sie das nicht, wird auf die Methode der `WebSecurityConfig` geguckt.

Falls die Authentifizierung failed, wird der User bzw Owner auf die Login Seite redirected und bekommt dazu eine Fehlermeldung. Eine Registrierung ist derzeit noch nicht möglich für den Owner, diesen muss man im Backend mittels einer Instanziierung erstellen.

3.2 Mobile Client

Der Client für die mobilen Endgeräte besitzt sechs verschiedene Screens/Views. Der Welcome-, Signin- und Signup-Screen sind in Abbildung 3.4 zu sehen. Die Funktion dieser Screens sollte durch ihre Namensgebung selbsterklärend sein. Beim klicken auf den Login- oder Signup-Button werden die Daten der Eingabefelder mittels POST-Request an die entsprechende API des Backends gesendet. Ob die Aktion erfolgreich war, wird durch ein Dialogfenster mitgeteilt. Bei erfolgreichem Login, wird der Empfangene JWT in einer Provider-Klasse gespeichert, der diesen der Applikation global zu Verfügung stellt.

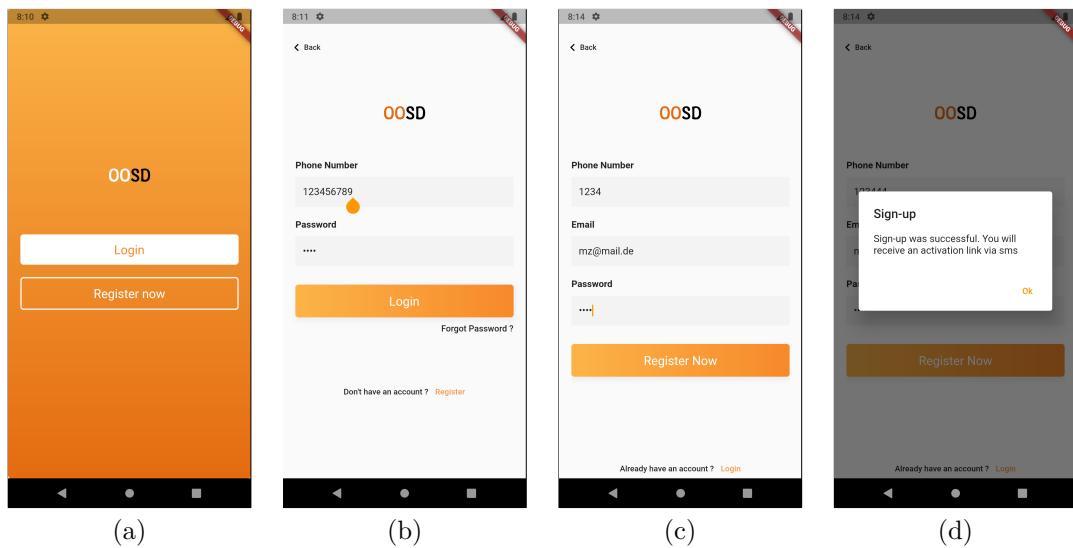


Abbildung 3.4: Welcome- (a), Signin- (b) und Signup-Screen (c&d)

Sobald der User erfolgreich angemeldet wurde, wird er auf den in Abbildung 3.5(a) zu sehenden Info-Screen weitergeleitet. Auf dieser Screen werden ihm alle verfügbaren Informationen über sich selbst angezeigt. Darunter auch in Textform mit farblicher Visualisierung, ob er gerade in einer Location eingecheckt ist oder nicht. Um die Infos vom Server abzufragen, wird die `ApiProvider`-Klasse verwendet. Diese stellt der App global Funktionen zur Verfügung um mit dem Backend zu kommunizieren. Dieser Provider wiederum nutzt für jede Anfrage den `AuthProvider` welcher den benötigten JWT-Header bereitstellt.

Es existiert ein Drawer (Abbildung 3.5(b), mit dem zwischen den Screens navigiert werden kann. Die Inhalte des Checkin- und Checkout-Screens werden je nachdem ob der User gerade eingechockt ist oder nicht dynamisch angepasst. Ist der User z. B. noch nirgends eingechockt wird beim wechseln in den Checkin-Screen sofort der QR-Scanner aktiviert. Ist er bereit eingechockt, wird ihm dies mitgeteilt und ein Button angeboten mit dem er auf den Checkout-Screen wechselen kann. Andersherum ist das ähnlich geregelt. Sobald der QR-Scanner einen QR-Coder erkennt, versuchter er den User über die API einzuchecken. Wenn das erfolgreich ist, wird er zum Info-Screen geleitet und sieht dort in grüner Schrift visualisiert, dass er eingechockt ist. Sollte es Probleme beim Einchecken geben, wird der User ebenfalls auf den Info-Screen geleitet, bekommt jedoch in einem Dialog eine Mitteilung, dass etwas schief gelaufen ist.

Durch das Klicken auf den Logout-Button im Drawer wird der User ausgecheckt, der JWT gelöscht und wieder auf den Welcome-Screen geleitet.

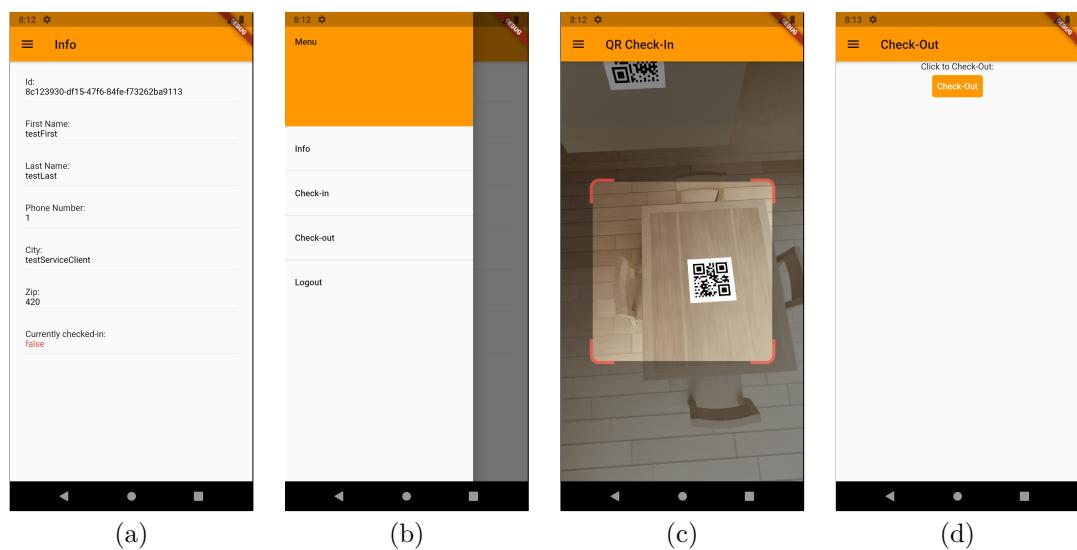


Abbildung 3.5: Info- (a), Checkin- (c) und Checkout-Screen (d). Drawer (b)

3.3 Web UI

Wie bereits in dem Unterkapitel über Vaadin beschrieben, ist hier die Besonderheit, dass das Frontend ebenfalls in Java geschrieben werden konnte. Es ist möglich, direkt über die Service Klassen auf die Datenbank zuzugreifen, ohne über die Controller zu gehen und Requests und Responses hin- und herzuschicken, wie im traditionellen Webflow.

FABQ OOSD

Login

Benutzername
owner

Passwort

Einloggen

Passwort vergessen?

Abbildung 3.6: Login Seite der WebUI

Die Startseite der Webapplikation ist ein Index mit den Lokalitäten des Besitzers. Diese Liste hat verschiedene Buttons, mit denen der Owner verschiedene Funktionalitäten aufrufen kann.

Neue Location hinzufügen						Logout		
ID	City	Max Visitors	Street	Zip	QR Code	aktuelle Besucher	Besuche	Löschen
1	LocCity	10	locationStreet	6969	QR Code	aktuelle Besucher	Besuche	Löschen
4	Augsburg	10	Augsburger Straß...	86157	QR Code	aktuelle Besucher	Besuche	Löschen

Abbildung 3.7: Startseite der Webapplikation

Der User kann für jede Lokalität den generierten QR-Code anzeigen lassen. Dieser Button ruft die Methode `getQRCodeForLocation()` der LocationService Klasse auf, die ein `BufferedImage` Objekt zurückgibt. Anschließend wird dieses in ein `Image` Objekt geparsed und in einem PopUp Dialog dargestellt, sodass man diesen an die Klienten weitergeben kann.

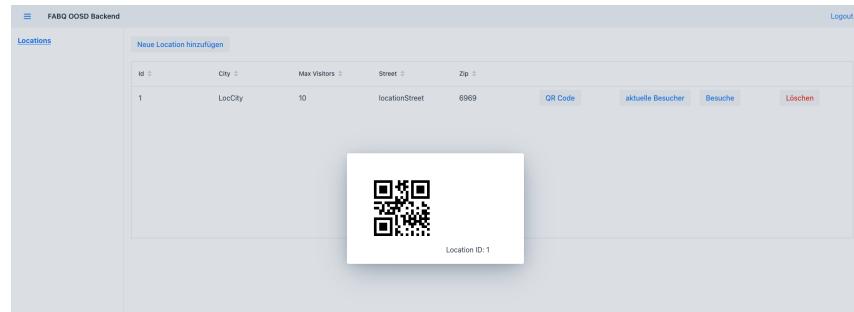


Abbildung 3.8: QR Code Anzeige

Eine weitere Funktionalität ist der Abruf der aktuellen Besucher. Derzeit wird beim Klick auf diesen Button ein PopUp Dialog getriggert, welches die Anzahl der eingekennzeichneten Besucher aufzeigt. Hier wird die Methode `getCurrentVisits()` der VisitService Klasse aufgerufen, die anhand der Lokalität die Besucher als eine Liste zurückgibt, die derzeit keine Check-Out Zeit besitzen.

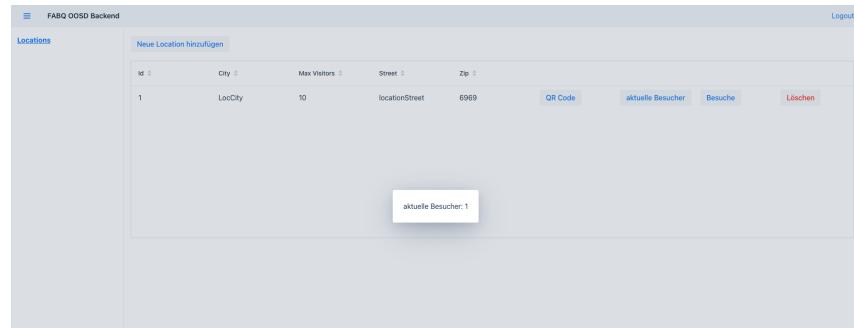


Abbildung 3.9: Aktuelle Besuche

Um die Besucher auch warnen zu können, falls in einem Zeitraum ein Coronafall aufgetreten ist, ist es erforderlich die Besucherliste auslesen zu können. Der Button `Besuche` steuert die Route `/visits/(id)` mit der ID der Location als URL Parameter an. Es öffnet sich eine Liste mit allen Klienten, die die Lokalität besucht haben. Dabei stehen der Name, die Kontaktdaten und der Zeitstempel der Check-In und Check-Outs des Besuchs drin.

Besucherhistorie					
Vorname	Nachname	Email	Handnummer	Check-in	Check-out
C1	C1Last	c@email.com	018054646	05.02.21, 18:12	

Abbildung 3.10: Besucherhistorie

Kapitel 4

Schritte zur Inbetriebnahme

In diesem Kapitel wird beschrieben, wie sich das Projekt auf dem lokalen Rechner starten lässt. Voraussetzung hierfür ist, dass Java (Version 11), Vaadin (Version 14.4.1), die aktuelle Flutter Version und eine aktuelle PostgreSQL Version installiert ist. Vom Vorteil ist es außerdem auch, wenn die IDE IntelliJ vorhanden ist, da man dadurch nicht Maven installieren muss. Falls dies nicht der Fall ist, muss außerdem noch Maven separat runtergeladen werden, um die Abhängigkeiten runterladen zu können.

Datenbank

Zuerst muss die Postgres Datenbank erstellt werden. Anschließend definiert man in der `application.properties` die Datenbankdaten wie den Usernamen und das Passwort und die URL, auf die die Datenbank erreichbar ist.

Backend/Vaadin

Das Projekt kommt mit ein `pom.xml`, die man mit Maven builden kann. IntelliJ bietet dafür ein Feature an, das die File automatisch erkennt und den Build Cycle durchläuft. Verwendet man VS Code dafür, gibt es ein Plug-In, der dies auch automatisch erkennen kann. Möchte man den Maven Build manuell vornehmen, kann man dies auch im Terminal anhand von Befehlen realisieren.

Um den Server anschließend zu starten, reicht es den Start Knopf im IntelliJ zu klicken, der den Default Tomcat Server auf Port 8080 laufen lässt. Alternativ lässt sich dies auch anhand des Befehls `mvn spring-boot:run` ausführen.

Hinweise zur Debug-Konfiguration: Standardmäßig ist der Server so konfiguriert, dass ein Mobile-User beim Registrieren sofort freigeschaltet wird und kein Token versendet wird. Dies kann in der Klasse `DebugConfig` angepasst werden. Die App wurde auf einen Emulator getestet und kommuniziert mit einem Localhost Server. Für die die Bestätigung der Telefonnummer wird jedoch ein echtes Telefon benötigt, das den Aktivierunglink mit dem Token empfängt. Diesen Link jedes Mal in den Emulator zu bringen war etwas umständlich, deshalb wurde nach erfolgreichen Tests die eben erwähnten Debug-Einstellungen hinzugefügt. In dieser Klasse kann auch die komplette Security deaktiviert werden.

Android Client

Zur Entwicklung und Debugging wurde ein Android Virtual Device verwendet. Dies wird auch für die Inbetriebnahme empfohlen. Der Emulator hatte folgende Spezifikationen:

Modell: Pixel 3a, API: 28, Target: Android 9.0, CPU: x86

Um die beigefügte APK zu installieren, genügt es bei einem AVD die APK-Datei per Drag and Drop auf den Emulator zu ziehen. Um den Sourcecode selbst zu Kompilieren ist eine aktuelle Flutter Version und all ihre Abhängigkeiten nötig. Diese können nach der Flutter Installation mit dem Kommandozeilenbefehl `flutter doctor` überprüft werden. Mit dem Terminal Befehl `flutter build apk --debug` im Ordner in dem sich die `pubspec.yaml`-Datei befindet (Root-Ordner des Flutterprojekts), kann dann die APK gebaut werden. Hierbei ist **wichtig**, dass dies ein **Debug**-Build ist. Bei Release-Builds wird von Android und iOS die Kommunikation mittels Https vorausgesetzt. Das Backend unterstützt jedoch nur Http. Bei Fehlern durch fehlende Pakete können diese durch den Befehl `flutter pub get` heruntergeladen werden.

Wichtiger Hinweis, um mit dem Backend zu kommunizieren: In dem Verwendeten Emulator ist die Localhost Adresse standardmäßig nicht 127.0.0.1 sondern 10.0.2.2. Falls diese beim verwendeten Emulator abweichen sollte, müssen die Localhost-IPs im Sourcecode mit der korrekten IP ersetzt werden. Um den QR-Scanner bzw. den Check-in im Emulator zu testen, können unter den Camera-Einstellungen des AVD-Emulators Bilder für die virtuelle Kamera hinzugefügt werden. Hierfür einfach die QR-Codes aus der Web-UI herunterladen und in den Einstellungen hinzufügen. In der virtuellen Welt muss die Kamera dann so bewegt werden, dass sie auf einen QR-Code ausgerichtet ist.

Seed Daten

Das Projekt führt in der main die `loadData()` Methode aus, die initiale Objekte erstellt. Daher kann anhand von folgenden Daten, die Applikation gestartet und benutzt werden.

1. Web UI:
 - Username: owner, Password: 1234
2. Mobile-Client:
 - Phonenummer: 1, Password: 1234
 - Phonenummer: 018054646, Password: 1234

Kapitel 5

Fazit

Im Rahmen dieses Projekts wurde eine Webapplikation zum Management von Klienten in einer Örtlichkeit zu Coronazeiten realisiert. Dabei konnten alle hoch priorisierten Features, wie das Erstellen eines QR-Codes zu jeder Lokalität oder das Einchecken mittels Mobile-App und QR-Code, umgesetzt werden. Das Projekt besitzt eine Web Oberfläche, in dem der Besitzer sich anmelden kann, um die Klienten einzusehen, die seine Lokalität oder Lokalitäten besucht haben.

Des Weiteren kann sich ein User mittels einer Telefonnummer über die Mobile Applikation registrieren und somit seine Identität bestätigen. Somit werden Falschinformationen vorgebeugt. Der User kann mit Einscannen des QR-Codes sich automatisch einchecken. Mithilfe eines Check-Out Buttons kann sein Besuch als beendet wieder registriert werden.

Lediglich weitere Anforderungen mit mittleren oder niedrigen Priorisierungen, wie die GoogleAuth Validierung des Kontos, konnten nicht erfüllt werden.

Mögliche Verbesserungen wären Validierungen zu der maximalen Besucherzahl und dass man als User sich nicht mehr einscannen kann, wenn eine bestimmte Anzahl bereits eingecHECKT ist. Für beide Frontend Implementierungen können die Form Input Felder um Validierungen ergänzt werden. Eine weitere Verbesserung wäre für die Web UI, dass die Besucherhistorie anhand eines Zeitraums gefiltert werden kann, um alle Klienten bei einem Fall schneller sortieren zu können. Ein Feature, das zu Beginn ebenfalls zur Debatte stand, ist die Implementierung von Notifications, also Benachrichtigungen. Hiermit soll möglich sein, auf die Mobile App Warnungen zu senden, falls der Klient potentiellen Kontakt zu einem Corona Fall hatte, etwa wenn diese zu einem Zeitpunkt beide in der Lokalität eingecHECKT waren.

Bei der Erstellung und Implementierung dieser Anwendung fiel unser persönlicher Eindruck sehr positiv aus.

Vaadin bietet eine gut strukturierte und einstiegsfreundliche Umgebung und ist somit auch für Neulinge in der Webentwicklung gut geeignet. Durch das von Beginn an umgesetzte Projektmanagement und die sofortige Dokumentation von Besprechungen, hat die Gruppenarbeit miteinander sehr gut harmoniert und funktioniert.

Quellenverzeichnis

Literatur

- [1] Rod Johnson u.a. „The spring framework–reference documentation“. In: *interface* 21 (2004), S. 27 (siehe S. 3).

Online-Quellen

- [2] *Flutter Architektur*. URL: <https://flutter.dev/docs/resources/architectural-overview> (besucht am 05.02.2021) (siehe S. 4).
- [3] *Flutter FAQ*. URL: <https://flutter.dev/docs/resources/faq> (besucht am 05.02.2021) (siehe S. 4).
- [4] *Flutter Website*. URL: <https://flutter.dev/> (besucht am 05.02.2021) (siehe S. 4).
- [5] *Hibernate Website*. URL: <https://hibernate.org/orm/> (besucht am 05.02.2021) (siehe S. 5).
- [6] *JDBC Wikipediaseite*. URL: https://de.wikipedia.org/wiki/Java_Database_Connectivity (besucht am 05.02.2021) (siehe S. 5).
- [7] *Lombok Website*. URL: <https://projectlombok.org/> (besucht am 05.02.2021) (siehe S. 6).
- [8] *Preise für den Darfichren Service*. URL: <https://darfichrein.de/preise> (besucht am 05.02.2021) (siehe S. 1).
- [9] *QR Code Scanner Package*. URL: https://pub.dev/packages/qr_code_scanner (besucht am 05.02.2021) (siehe S. 5).
- [10] *Skia Rendering Bibliothek*. URL: <https://skia.org/> (besucht am 05.02.2021) (siehe S. 4).
- [11] *Website der Darfichrein GmbH*. URL: <https://darfichrein.de/> (besucht am 05.02.2021) (siehe S. 1).
- [12] *Website IntelliJ*. URL: <https://www.jetbrains.com/de-de/idea/> (besucht am 05.02.2021) (siehe S. 6).
- [13] *Website Postman*. URL: <https://www.postman.com/> (besucht am 05.02.2021) (siehe S. 7).
- [14] *Website Vaadin*. URL: <https://vaadin.com/> (besucht am 05.02.2021) (siehe S. 4).
- [15] *Website Visual Studio Code*. URL: <https://code.visualstudio.com/> (besucht am 05.02.2021) (siehe S. 6).
- [16] *Website von VS Code*. URL: <https://code.visualstudio.com/> (besucht am 05.02.2021) (siehe S. 5).
- [17] *Zxing Website*. URL: <https://opensource.google/projects/zxing> (besucht am 05.02.2021) (siehe S. 5).