Entwicklung einer Go-React Applikation für ein Büroplatz-Buchungssystem

Vincent Bärtsch

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Rhein-Main

17.08.2021

Betreuer

Prof. Dr.-Ing. Ludger Martin, Hochschule Rhein-Main

Christoph Pascher, Browserwerk GmbH

Erklärung gem. ABPO, Ziff. 4.1.5.4

Ich versichere, dass ich die Bachelor-Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Mainz, 09.08.2021
Ort, Datum

Bärtsch
Unterschrift Studierende/Studierender

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Arbeit:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger	x	
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	x	
Veröffentlichung des Titels der Arbeit im Internet	x	
Veröffentlichung der Arbeit im Internet	x	

Mainz, 09.08.2021
Ort, Datum

Bärtsch
Unterschrift Studierende/Studierender

Abstract

Entwicklung einer Go-React Applikation für ein Büroplatz-Buchungssystem

Im Rahmen dieser Bachelorarbeit wurde eine Büroplatz-Buchungsapplikation entwickelt. Die hier entwickelte Buchungsapplikation unterstützt Unternehmen dabei, das hybride Arbeiten umzusetzen. Der Fokus lag darin, eine nutzungsfreundliche Umgebung zu bauen, in der man als Nutzer Reservierungen eines Arbeitsplatzes vornehmen kann und diese dann geordnet angezeigt bekommt. Zuerst wurden die funktionalen und nichtfunktionalen Anforderungen erstellt. Anhand dieser wurden im Folgenden die Endpunkte und Datenbankmodelle aufgebaut. Für die API wurde die Programmiersprache Go und als nicht relationale Datenbank, MongoDB genutzt. Für das Frontend wurde die Bibliothek React verwendet und Typescript zu Hilfe gezogen. Das Ein- und Ausloggen wird mit einem JSON Web Token geregelt. Ein Admin-Benutzer kann die Seite mit neuen Räumen und Arbeitsplätzen pflegen. Hierfür wurde im Frontend ein Baukasten für einen Platzplaner erstellt. In diesem hat man die Möglichkeit, die Arbeitsplätze wie im Büro darzustellen. Ziel der vorliegenden Bachelorarbeit war es, eine benutzerfreundliche Buchungsplattform auf den modernen Endgeräten zu konzipieren. Das Frontend ist gut gelungen. Gerade das Suchen von Räumen und das Erstellen von übersichtlichen Arbeitsplatzplänen, erleichtern die alltägliche Reservierung für den Nutzer. Aufgrund des unabhängigen Back- und Frontends lässt sich die Applikation für jedes Unternehmen individualisieren und erweitern. Um einen hohen Qualitätsstandard zu erreichen, wäre die Durchführung von automatischen Tests im Frontend und im Backend nötigt gewesen. Dieser Gesichtspunkt ergabe sich jedoch erst im Laufe der Arbeit, so dass es sinnvoller gewesen wäre, von Anfang mit Tests geplant zu haben. Dann wäre schneller aufgefallen, dass der Quellecode im Backend anders aufgebaut werden muss, um automatische Tests durchzuführen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Im März 2020 werden aufgrund der Corona-Pandemie Großveranstaltungen und Gottesdienste verboten, es werden bis auf die Supermärkte fast alle Geschäfte geschlossen, sogar die Schulen und Kitas sind zu. Landes- und sogar weltweit wurden viele Unternehmen mit der Situation konfrontiert, auch erstmalig, Homeoffice einzuführen, was dann im Verlauf der Pandemie während des Jahres 2020 schnell zur Regel wurde.

Diese Erfahrungen haben uns gezeigt, dass das Homeoffice eine gute Möglichkeit ist, die Arbeit ohne weitere Probleme nach Hause zu verlagern. Erstaunlich ist, dass vor der Industrialisierung so gut wie alle Menschen von zu Hause gearbeitet haben. Um so mehr wundern sich Müller et al. [MTI21], dass das Arbeiten im Homeoffice heutzutage ein Privileg ist. Aber wie wird es in der Zukunft aussehen? Werden wir nach der Pandemie alle wieder jeden Tag vom Büro aus arbeiten?

Eine Studie vom 17. Dezember 2020 des Wirtschaftsprüfungs- und Beratungsunternehmens EY [Deu20] sagt aus, dass 76 Prozent aller Befragten eine Mischung aus Büro und Remote-Work bevorzugen. Ein Grund dafür ist, dass eine klare Trennung von Arbeits- und Privatleben gewünscht wird. Viele Arbeitnehmer wollen aber auch nicht auf die sozialen Kontakte mit Kollegen verzichten.

Hier entsteht der Ansatzpunkt für das hybride Arbeiten. Dies ist eine Mischung aus Homeoffice und Büroarbeit. In diesem Modell kann sich der Arbeitnehmer selbst aussuchen, wann er ins Büro kommen und wann er von zu Hause arbeiten möchte. Dies bedeutet für den Einzelnen ein hohes Maß an Flexibilität. Es ist aber auch sehr

attraktiv für den Arbeitgeber: Bei einem solchen hybriden System benötigt nicht jeder Angestellte immer einen Arbeitsplatz vor Ort. Dies hat zur Konsequenz, dass sich die Bürofläche verkleinern lässt und somit Kosten eingespart werden.

Auch die Browserwerk GmbH ist während der Corona-Pandemie nach und nach vollständig auf Remote umgestiegen. Sobald sich die Situation wieder ändert, soll hier ein hybrider Arbeitsplan angeboten werden.

1.2 Aufgabe

Damit immer eine ausreichende Anzahl von Arbeitsplätzen vorhanden ist, wird eine passende Buchungs-Applikation benötigt. Es soll ein Backend entwickelt werden, das Daten verwalten soll. Auch wird erarbeitet, wie die Daten in einer Datenbank am besten abgespeichert werden können. Außerdem wird ein Authentifikationssystem benötigt, um herauszufinden, welcher Mitarbeiter wann welchen Arbeitsplatz gebucht hat. Zur Koordinierung und Verteilung von Berechtigungshierarchien wird ein Rollenmanagement realisiert, welches sich zum einen auf die Mitarbeiter- und zum anderen auf die Admin-Rolle beschränken soll.

Das Frontend sollte auf allen aktuellen mobilen Geräten und in den aktuellen Browsern erreichbar sein. Die Mitarbeiter sollen die Möglichkeit besitzen, sich selbstständig in das System einzuloggen. Nach dem Einloggen können sie über die Plattform sehen, welche Arbeitsplätze frei und welche belegt sind. Auch sollen sie Arbeitsplätze für einen bestimmten Tag reservieren oder wieder freigeben können. Die Rolle des Admins bietet die Möglichkeit, flexibel neue Arbeitsplätze zu erstellen oder zu entfernen.

Diese wird so allgemein gehalten wie möglich, sodass die Applikation von jedem Unternehmen genutzt werden könnte, die ein solches Arbeitsprinzip anstreben möchte.

Kapitel 2

Grundlagen

Im Grundlagen-Kapitel werden alle Begriffe erklärt, die in der Arbeit verwendet werden. Zudem wird ein Blick auf den aktuellen Markt geworfen und es werden drei vergleichbare Produkte beschrieben.

2.1 Begriffe

In diesem Abschnitt werden die Begriffe näher erklärt und definiert, die in der Arbeit verwendet werden.

2.1.1 Representational State Transfer

Der Architekturstil, der dem Web zugrunde liegt, heißt Representational State Transfer oder einfach REST. Dass REST die Grundlage der modernen Web-Architektur wurde, erklärt Jakl [Kam+14] damit, dass das System die Anforderungen von verteilten Hypermedia-Systemen im Internet erfüllt. Dies wird durch Skalierbarkeit, Allgemeinheit der Schnittstellen, unabhängige Bereitstellung und die Ermöglichung zwischengeschalteter Komponenten zur Reduzierung der Netzwerklatenz erreicht. REST wurde 2000 von Roy Fielding an der University of California, Irvine, entwickelt. Hierbei ging es um die Philosophie des Open Web. Heute beschreibt es Battle [BB08] als die moderne Alternative zu SOAP. Während SOAP den Ansatz für Webdienst vollwertige Remote-Objekten mit Methodenaufruf und gekapselter Funktionalität verwendet, nutzt REST nur Datenstrukturen mit ihren Übertragungszuständen.

Trotz alledem ist REST kein Standard, sondern nur eine Reihe von Randbedingungen, erklärt Patni [Pat17]. Allerdings bedient sich REST an einigen Standards wie zum Beispiel an HTTP oder JSON. Es ist auch nicht an HTTP gebunden, wird aber häufig in diesem Zusammenhang verwendet.

Eine typische RESTful-Anfrage benötigt eine von vier Standard-HTTP-Methoden: GET, PUT, POST und DELETE. Dazu noch eine Domain-Adresse des API-Servers, den Namen der API-Methode. Das Format der Rückgabedaten und die Parameter können auch hinzugefügt werden. Hierbei ist Domain plus Methodenname eine Ressourcen-URL.

Methoden können idempotent und nicht-idempotent sein. Als Idempotent bezeichnet man Operationen, wenn der Client denselben Aufruf wiederholt ausführt und immer dasselbe Ergebnis erzielt, beschreibt Dazer [Daz12] in seiner Arbeit.

Mit der GET-Methode bekommt man Ressourcen zurück, die auf dem Server angefragt wurden. Diese Methode darf keine Änderungen durchführen und ist so sicher und idempotent, da sie keinen Einfluss auf die Daten/Datenbank hat. Wenn man eine Ressource erstellen möchte, benutzt man die POST-Methode, die nicht-idempotent ist, da man die Methode N-mal aufrufen könnte und so N neue Ressourcen erstellen würde. Aber auch mit der PUT-Methode kann man Ressourcen erstellen, wobei diese dann zusätzlich überprüft, ob bereits solche vorhanden sind. Wenn dies der Fall sein sollte, wird die Ressource nur verändert; falls nicht, wird sie neu erstellt. Dies bedeutet, dass bei mehrfacher Anfrage die Ressource überschrieben wird, also effektiv nichts geändert. Daher ist PUT idempotent. Abschließend ist noch auf die DELETE-Methode hinzuweisen, die Ressourcen löscht. Diese ist ebenfalls idempotent, da die Ressource bei dem ersten Aufruf gelöscht wird. Bei den nächsten Aufrufen gibt es eine Fehlermeldung, die anzeigt, dass die Ressource nicht mehr vorhanden ist.

2.1.2 Go

Go wurde von Google-Mitarbeitern entwickelt und am 10. November 2009 zu einem öffentlichen Open-Source-Projekt [Goo09]. Dies bedeutet, dass es noch eine sehr junge Sprache ist. Aber dennoch ist sie schon jetzt recht beliebt und vor allem bekannt für ihre hervorragende Leistung bei der nebenläufigen Programmierung.

Sogar Google selbst greift oft auf seine eigene Sprache zurück und entwickelt verschiedene Projekte mit ihr, wie zum Beispiel den Download-Server von Google.

Die junge Programmiersprache Go gehört zum größten Teil zu der C-Familie. Dies bedeutet, dass sie eine kompilierbare Programmiersprache ist. Auch hat Go einen bedeutenden Input aus der Pascal-/Modula-/Oberon-Familie, wie sie selber auf ihrer Seite beschreiben [GoL21]. Aber auch aus anderen Sprachen kommen einige Inspirationen. Man könnte somit sagen, Go verwendet die guten Aspekte aus den vorhandenen Programmiersprachen und fügt sie zu einer Sprache zusammen.

Der Hauptvorteil von Go liegt in der Geschwindigkeit und der Fehlererkennung bei der Kompilierung, beschreiben Yellavula und Naren [Yel17]. Verschiedene Benchmarks haben bewiesen, dass Go in Bezug auf die Rechenleistung schneller ist als dynamische Programmiersprachen.

2.1.3 MongoDB

MongoDB ist eine dokumentbasierte Datenbank. Sie ist hauptsächlich in C++ programmiert und gehört zu den führenden NoSql-Datenbanken. Ploetz, Kandhare et al. [Plo+18] erklären, dass durch das JSON-ähnliche Dokumenten-Schema eine hohe Leistung, Verfügbarkeit und einfache Skalierbarkeit bereitgestellt wird. Durch die Datenbank-Partitionierung lassen sich Datenbanken einfach skalieren und die Daten auf mehrere Systeme verteilen.

Die Daten werden in Collections gespeichert. Eine Sammlung ist schemalos, so können Dokumente unterschiedliche Feldersätze besitzen. Trotz allem sollen nur verwandte Dokumente in eine Collection gespeichert werden.

2.1.4 React

React ist eine JavaScript-Bibliothek und wurde ursprünglich von Ingenieuren bei Facebook entwickelt, um die Herausforderungen bei der Entwicklung komplexer Benutzeroberflächen mit Datensätzen, die sich im Laufe der Zeit ändern, zu lösen, beschreiben Gackenheimer und Cory [Gac15] in ihrem Buch. Hierbei arbeitet React deklarativ, sobald Daten erneuert werden, wird die passende Komponente angepasst und neu gerendert. Dadurch, dass man seinen Code in Komponenten aufteilt, werden Wartbarkeit oder Erweiterung erleichtert.

2.1.5 TypeScript

TypeScript ist eine Erweiterung von JavaScript, die eine einfache Entwicklung von umfangreichen JavaScript-Anwendungen ermöglicht. Es bietet ein Modulsystem, Klassen, Schnittstellen und ein Typsystem. Der TypeScript-Compiler prüft das Programm und gibt JavaScript zurück, um es im Browser zu verwenden. Dadurch entsteht der Vorteil, dass in TypeScript alle JavaScript-Funktionen kompatibel sind.

2.1.6 JSON Web Token (JWT)

JSON Web Token (JWT) ist ein offener Standard (RFC 7519) [Aut21], der eine kompakte und in sich geschlossene Methode zur sicheren Übertragung von Informationen zwischen Parteien als JSON-Objekt definiert. JWTs können mit HMAC-Algorithmus verschlüsselt oder mit einem öffentlichen/privaten Schlüsselpaar mit RSA oder ECDSA signiert werden. Am häufigsten wird JWT zur Autorisierung genutzt, da es einen geringen Overhead hat und einfach über verschiedene Domänen hinweg verwendet werden kann [Med20]. JSON Web Token besteht aus den drei Teilen Header, Payload und Signatur, welche im Token mit Punkten getrennt werden.

2.1.7 bcrypt-Algorithmus

Bcrypt ist ein Kennwort-Hash-Verfahren, das auf der Blowfish-Blockchiffre basiert. Dieser Algorithmus wurde entwickelt, um resistent gegen Brute-Force-Angriffe zu sein, erklären Malvoni, Designer, Knezovic [MS14]. Zudem erleichtert der Algorithmus das Verschlüsseln und Speichern von Passwörtern.

2.2 Vergleichbare Produkte

Auf dem Markt gibt es viele Arbeitsplatz-Buchungssysteme. Vor allem seit der Corona-Pandemie ist die Nachfrage stark gestiegen. In diesem Abschnitt sollen verschiedene Systeme genauer betrachtet werden. Alle sind unterschiedlich in der Anwendung; gemeinsam haben sie, dass sie nur entgeltlich zu verwenden sind. Im Ergebnis sind alle gleich: Man kann Arbeitsplätze damit verwalten.

2.2.1 Arboo

Das Arboo Buchungssystem zeichnet sich dadurch aus, dass es sich mit Microsoft verbinden kann [arb21]. So ist es möglich, seinen Arbeitsplatz in den Office 365-Programmen wie z. B. Teams, Outlook-Kalender oder über eine SharePoint Online-Seite, zu buchen. Das System zeichnet durch eine schnelle Arbeitsplatz-Verwaltung aus. Man kann damit in der Corona-Zeit schnell auf die Anforderungen der Gesundheitsbehörden reagieren, zum Beispiel garantierte Abstände zwischen den Mitarbeitern, da sich leicht Grundrisse von Räumen erstellen lassen.

2.2.2 Flexopus

Flexopus bietet kein reines Arbeitsplatz-Buchungssystem; auch Parkplätze und Besprechungsräume können reserviert werden [fle]. Mit einer Mobilen– oder einer Desktop-App lassen sich einfach Plätze buchen. Diese können vom Admin ganz einfach blockiert werden. Auch ist der Admin in der Lage, ein Rechtemanagement anzulegen, sodass nicht jeder jeden Platz reservieren kann. Außerdem können Ausstattungen zu Arbeitsplätzen hinzugefügt werden. Der Nutzer kann Arbeitsplätzen filtern, die nur eine bestimmte Ausstattung haben. Ein weiterer Vorteil ist die Belegungsplan-Ansicht. Hier kann der Nutzer live nachschauen, wann welcher Platz im Raum belegt ist. Auch wird von Flexopus behauptet, dass alles in Deutschland gehostet wird und es zu 100 Prozent DSGVO-konform ist.

2.2.3 Condecosoftware

Das Buchungssystem Condecosoftware legt den Fokus eher auf die Hardware [Ltd]. Es bietet ein Arbeitsplatz-Display, auf dem man den Arbeitsplatz leicht buchen kann. Der Vorteil ist, dass man nur eine RFID-Zugangskarte braucht, um sich einzuloggen. Auch bietet das System eine Mitarbeitersuche. Man kann demzufolge im System sofort sehen, wo sich die einzelnen Mitarbeiter eingeloggt haben.

Kapitel 3

Anforderungen

Die Aufgabe dieser Bachelorarbeit besteht darin, eine Web-Applikation zu entwickeln, mit der man Arbeitsplätze reservieren und verwalten kann. Die Anwendung soll auf allen aktuellen Mobilen Geräten und Browsern erreichbar sein.

3.1 Funktionale Anforderungen

Es wird unterschieden zwischen ohne Account = none, mit aktiviertem Account = Member, und Account mit Administrationsrechten = Admin. Der Nutzer ohne Account hat zunächst nur die Möglichkeit, sich einen Account zu erstellen [FA101]. Danach kann der Account nur von einem Admin aktiviert oder gelöscht werden [FA113].

Sobald der Account aktiviert wurde, wird der Nutzer zu einem Member. Der Member soll die Möglichkeit haben, sich ein- und auszuloggen [FA102]. Auch kann er sich alle Räume, die zur Nutzung zur Verfügung stehen, anzeigen lassen [FA103]. Um sich dann eine Raum genau anzuschauen, wird eine Detail-Ansicht gebraucht [FA104]. Auch soll es eine Möglichkeit geben, nach einem Raumnamen zu suchen [FA105]. In den Räumen sollen sich Raumpläne befinden, die alle Arbeitsplätze im Raum anzeigen [FA106]. Der Member braucht jetzt eine Möglichkeit, den Raum zu reservieren oder eine bereits von ihm getätigte Reservierung zu löschen [FA107]. Auch muss man die Reservierung eines Raumes anzeigen lassen [FA108]. Außerdem soll es eine Möglichkeit geben, in einer Übersicht alle getätigten Reservierungen eines Nutzers zu sehen [FA109].

Funktionale Anforderungen (FA)	**Beschreibung**
none	
FA101	Account erstellen
Member	
FA102	Account einloggen/ausloggen
FA103	Liste von Räumen
FA104	Raum anzeigen
FA105	Nach Raum suchen
FA106	Raumplan anzeigen
FA107	Reservierung erstellen/löschen
FA108	Alle Reservierungen eines Arbeitsplatzes anzeigen
FA109	Alle Reservierungen des Accounts anzeigen lassen
Admin	
FA110	Raum erstellen/löschen/bearbeiten
FA111	Raumplan erstellen/bearbeiten
FA112	Arbeitsplätze erstellen/löschen/bearbeiten
FA113	Account aktivieren/löschen
FA114	Account anpassen
FA115	Alle Accounts anzeigen
FA116	Nach Account suchen

Tabelle 3.1: Auflistung der funktionalen Anforderungen

Um die Applikation zu verwalten, benötigt man einen Admin. Dieser soll Räume erstellen, bearbeiten und löschen können [FA110]. In diesen Räumen kann er einen Raumplan erstellen oder bearbeiten [FA111]. Dabei kann der Admin Arbeitsplätze erstellen, bearbeiten und löschen [FA112]. Damit er alle Accounts bearbeiten [FA114] kann, benötigt er noch eine Seite, auf der alle Accounts aufgelistet werden [FA115]. Damit das Suche von Accounts erleichtert wird, soll es eine Möglichkeit geben, die Accounts mit einem Suchfeld zu filtern [FA116].

3.2 Nichtfunktionale Anforderungen

Aus den funktionalen Anforderungen ergeben sich die nichtfunktionalen Anforderungen. Zunächst sollen Backend und Frontend unabhängig voneinander aufgebaut sein [NF101], sodass man die einzelnen Systeme einfach ersetzen und auf verschiedenen Servern laufen lassen könnte. Ein weiterer wichtiger Punkt ist hier die Handhabung und Benutzbarkeit [NF102]. Die Applikation soll selbsterklärend und in-

Nichtfunktionale Anforderungen (NF)	Beschreibung
NF101	Backend und Frontend unabhängig
NF102	Handhabung und Benutzbarkeit
NF103	Effiziente Wartbarkeit
NF104	Einhaltung von Standards
NF105	Datensicherheit

Tabelle 3.2: Auflistung der nichtfunktionalen Anforderungen

tuitiv verwendbar sein, da Nutzer aller Alters- und Wissensgruppen das System nutzen. Erforderlich ist eine effiziente Wartbarkeit [NF103], um schnell auf Fehler oder Wünsche eingehen zu können. Dazu gehört auch, dass die Standards eingehalten [NF104] werden. Im Bereich der Datensicherheit [NF105] ist besonders wichtig, dass keine Daten weitergegeben werden. Wichtigste Daten sind nur für den Admin sichtbar. Als Unbefugter soll man keine Daten erhalten können.

Kapitel 4

Konzept

Um die Applikation zu erstellen, werden eine Datenbank, ein Backend und ein Frontend benötigt. Als Datenbank wird das dokumentenorientierte NoSQL-Datenbankmanagementsystem MongoDB verwendet. Für das Frontend wird die JavaScript-Bibliothek React genutzt. Damit Datenbank und Frontend kommunizieren können, wird ein RestAPI gebraucht, welches mit Go entwickelt wird.

4.1 Datenbank

Für die Datenbank wird eine MongoDB aufgesetzt, die eine NoSQL-Datenbank ist. Einer der Vorteile einer MongoDB ist, dass sie kein festes Schema besitzt, so lässt sich die Applikation später einfacher weiterentwickeln. Jederzeit können weitere Felder zu einer Collection hinzugefügt werden. Dies könnte in einer relationalen Datenbank aufwändiger sein, im schlimmsten Fall müsste eine komplette Umstrukturierung gemacht werden. Ein weiter Vorteil ist, dass die Daten auch ohne Probleme auf verschiedene Server verteilt werden könnten, was in der Dokumentation von MongoDB [Inc21] beschrieben wird. Dadurch könnte die Applikation eine erhebliche Anzahl an Räumen, Benutzer und Reservierungen speichern. Auf diesen Aspekt wird in dieser Arbeit allerdings nicht mehr eingegangen, weil das den Rahmen sprengen würde.

4.1.1 Datenbank-Schema

Da es eine NoSQL Datenbank ist, gibt es keine richtige Referenz. Die Collections stehen alle für sich. Jedes Datenobjekt bekommt eine UUID und wird als id gespeichert.

```
officeDb: schema
    + collections
        users: collection
            _id: String(0)
            email: String(0)
            name: String(0)
            password: String(0)
            role: String(0)
        reservations: collection
            _id: String(0)
            end_date: Long(0)
            room_name: String(0)
            start_date: Long(0)
            room_uuid: String(0)
            user_uuid: String(0)
            workspace_name: String(0)
            workspace_uuid: String(0)
        rooms: collection
            _id: String(0)
            description: String(0)
            name: String(0)
            specification: list(0)
            workspaces: list(0)
            specification.name: String(0)
            workspaces._id: String(0)
            specification.position_x: Double(0)
            workspaces.description: String(0)
            specification.position_y: Double(0)
            workspaces.name: String(0)
            specification.rotate: Double(0)
            workspaces.position_x: Double(0)
            workspaces.position_y: Double(0)
            workspaces.rotate: Double(0)
```

Listing 4.1: Datenbank-Schema

Um einen Account zu erstellen [FA101] wird eine User-Collection benötigt mit Name, E-Mail, Passwort und einer Rolle, welche string Objekte sind.

In der Reservations-Collection werden alle Reservierungen der Arbeitsplätze eingestellt [FA105]. Eine Alternative hätte auch sein können, die Reservierungen in die Room-Collection mit hinein zu speichern. Wenn man allerdings eine Reservierung von einem Account sucht, müsste man einmal durch alle Räume gehen, um diesen zu finden. Der Aufwand für die Datenbank würde dabei in keinem Verhältnis zum Nutzen stehen und würde eine Verwendung auf mobilen Geräten, auf denen schnell

alle Daten angezeigt werden müssen, verhindern. In der Reservations-Collection soll die Start- und Endzeit der Reservierung gespeichert werden.

Um den Raum und den Arbeitsplatz später auf den Reservierungs-Seiten einfacher anzuzeigen, sollen jeweils der Name und die UIDD zusätzlich gespeichert werden. So entstehen doppelte Daten, aber man erleichtert das Zugreifen auf die Namen, da NoSQL-Datenbank keine Joins unterstützen. Trotzdem ist es ein heikles Thema, da die Daten so schnell inkonsistent werden können. Sobald ein Name von einem Raum oder Arbeitsplatz verändert werden würde, müsste man auch in der Reservierungs-Collection alle Reservierungen mit dem Namen bearbeiten. Aber da es unwahrscheinlich ist, dass sich ein Name von einem Arbeitsplatz oder Raum ändert, soll es komplett verhindert werden. Sollte man sich entscheiden, die Namen zu ändern, müsste man den Raum oder Arbeitsplatz zuerst löschen und diese neue erstellen. Bei einem Löschvorgang sollte man trotzdem die gelöschten Elemente überprüfen und alle Reservierungen, die im Zusammenhang stehen, löschen.

Um Räume zu speichern und zu bearbeiten [FA109] wird eine Room-Collection benötigt. Name und Beschreibung des Raumes sind strings. Zudem sind noch Arrays von Arbeitsplätzen und Spezifikationen zu speichern. Die Spezifikationen sind später wichtig für die Arbeitsplatzplanung. In dieser werden nur der Name, die X-Position, Y-Position und Rotation gespeichert. Der Arbeitsplatz ist genauso aufgebaut, er bekommt nur noch eine Beschreibung.

4.2 Representational State Transfer Application Programming Interface

Die REST-API orientiert sich stark an den funktionalen Anforderungen. Jede Anforderung hat mindestens einen Endpunkt. Alle Endpunkte sind noch einmal in der Tabelle 4.1 aufgelistet. Damit eine Versionsverwaltung in der API vorgenommen werden kann, soll die erste Version unter `/api/v1/` erreichbar sein. Bei der Erstellung von der REST-API muss auf die Idempotente geachtet werden, damit es nicht zu Fehlern kommt.

Um einen Account zu erstellen [FA101] wird ein Endpunkt gebraucht, der eine POST-Methode erwartet. Dieser soll unter `/user` angesprochen werden. Übergeben werden sollen Name, E-Mail, und das Passwort.Hierbei muss beachtet werden, dass die E-Mail unique bleibt, da die E-Mail für den Login benötigt wird. Das bedeutet,

dass vor dem Erstellen geprüft werden muss, ob die E-Mail schon vergeben ist. So wird der Endpunkt zusätzlich Idempotent, obwohl es eine POST-Methode ist.

Damit der Nutzer sich nach Erstellung des Accounts ein- und ausloggen [FA102] kann, benötigt er hier zwei Endpunkte. Zum einen `/login` als POST-Anfrage und zum anderen `/logout` als DELETE-Anfrage. Beim Login müssen die E-Mail und das Passwort mitgegeben werden. Wenn der Login erfolgreich war, sollen die Account-Daten zurückgegeben werden, also Name, E-Mail und Rolle. Zudem soll ein Cookie erstellt werden mit einem JSON Web Token, womit die ganze Authentifizierung laufen soll. Dazu mehr im Kapitel 5. Der Login wäre dadurch nicht Idempotent. Würde man sich mehrmals hintereinander einloggen, würde sich der JWT immer wieder aktualisieren - was aber auch zu keinem Fehler führen würde. Aber Logout ist Idempotent, da man nichts löschen kann, was bereits gelöscht ist.

Der Endpunkt für die Raumliste muss alle Räume zurückgeben [FA103] und eine Möglichkeit für eine Seitennummerierung bieten. Mit einer GET-Methode und dem Pfad `/rooms` sollen alle Räume zurückgegeben werden. Da aber nicht alle Daten gebraucht werden, können die Arbeitsplätze und die Spezifikation wegfallen. Für die Seitennummerierung müssen Parameter wie „`page`“ und „`size`“ berücksichtigt werden. Dazu muss der Endpunkt die Seitenzahlen und die aktuelle Seitenzahl zu den Räumen zurückgeben.

Um einen bestimmten Raum anzeigen zu lassen, wird ein Endpunkt gebraucht, der eine GET-Methode erwartet [FA104]. Unter `room/[:uuid]` soll der Raum zurückgegeben werden.

Um nach einem Raum suchen zu können, muss der Raumlisten-Endpunkt angepasst werden [FA105]. Mit Hilfe eines weiteren Parameters „`search`“, in dem man den Namen des Raumes übergeben kann, soll eine Suche entwickelt werden.

Um einen bestimmtem Raumplan anzuzeigen [FA106], wird kein extra Endpunkt benötigt, da der Plan mit den Arbeitsplätzen im Raum gespeichert wird. Hier kann einfach der Raum-Endpunkt genutzt werden.

Zur Erstellung einer Reservierung [FA107] wird erneut ein Endpunkt mit einer POST-Methode benötigt. Dieser ist unter `/reservation` erreichbar. Ein Reservierungsobjekt muss mit einer Raum UUID, Arbeitsplatz UUID, Startzeit und Endzeit mitgegeben werden. Der Endpunkt ist nicht Idempotent. Da es aber wenig Sinn macht, dass jemand einen Arbeitsplatz zweimal reserviert, muss das überprüft und verhindert werden.

Um die Reservierung wieder zu löschen, wird ein weiterer Endpunkt gebraucht, der eine DELETE-Methode erwartet. Dieser muss mit `/reservation/[:uuid]` erreichbar sein. Jede Reservierung muss eine eigene UUID bekommen, um mit ihr schnell arbeiten zu können.

Damit der Benutzer sehen kann, wann welcher Arbeitsplatz frei ist, wird ein Endpunkt benötigt, der die Reservierungen anzeigen kann [FA108]. Mit einer GET-Methode unter `/reservation/[:uuid]` bekommt man alle Reservierungen eines Arbeitsplatzes zurück. Zum Filtern nach Zeit sind als Parameter Startzeit und Endzeit zu übergeben.

Um die Reservierung eines Accounts darzustellen [FA109], wird wieder eine GET-Methode gebraucht. Der Endpunkt ist unter `/user/reservierung/[:uuid]` erreichbar. Die UUID wird benötigt, um in der Datenbank die Reservierung vom Benutzer zu filtern. Zusätzlich soll man auch hier eine Zeitspanne als Parameter angeben können.

Darüber hinaus gibt es auch Endpunkte, die nur der Admin ausführen darf. Hierfür wird eine Middleware gebraucht, die die Rechte überprüfen kann, damit die Sicherheit der Daten gewährleistet wird [NF105].

Um einen Raum zu erstellen oder zu löschen, werden zwei Endpunkte gebraucht [FA110]. Um einen Raum zu erstellen, wird eine POST-Methode benötigt. Diese soll unter /room erreichbar sein und erwartet ein Raumobjekt. Der Endpunkt ist nicht Idempotent. Würde man öfters denselben Raum erstellen, würde das auch funktionieren, da erst das Backend die UUID setzen soll. So würde der Raum einfach mehrmals erstellt werden, aber dies würde zu keinem Fehler führen, da die UUIDs verschieden wären. Um einen Raum dann wieder zu löschen, wird eine DELETE-Methode gebraucht, die über /room/[:uuid] erreichbar ist.

Der Raumplan wird keinen zusätzlichen Löschen-Endpunkt brauchen, da die Arbeitsplätze im Raum gespeichert sind. Um dort etwas zu erstellen oder zu bearbeiten, muss man nur den Arbeitsplatz Array anpassen [FA111/FA112]. Dafür wird eine PUT-Methode benötigt, die unter /room/[:uuid] erreichbar ist. Dieser Endpunkt ist automatisch Idempotent, da hier nur Daten geupdated werden, was bedeutet, dass bei demselben Aufruf immer dasselbe passieren würde.

Um einen Account zu aktivieren, muss der Admin eine Rolle verteilen [FA113]. Auch dafür wird wieder eine PUT-Methode benötigt. Unter /user/[:uuid]/role soll ein Benutzer-Objekt mit UUID und einer Rolle übergeben werden. Um einen Ac-

Beschreibung	URI	Methode
Benutzer erstellen	/user	POST
Login	/login	POST
Logout	/logout	DELETE
Raumliste holen	/rooms	GET
Bestimmten Raum holen	/room/uuid	GET
Reservierung erstellen	/reservation	POST
Reservierung löschen	/reservation/uuid	DELETE
Reservierungen eines Arbeitsplatzes	/reservation/[:workspaceuuid]	GET
Reservierungen eines Accounts	/user/reservierung/[:useruuid]	GET
Raum erstellen	/room	POST
Raum löschen	/room/[:uuid]	DELETE
Raumplan updaten	/room/[:uuid]	PUT
Benutzer Rolle ändern	/user/role	PUT
Benutzer löschen	/user/[:uuid]	DELETE
Benutzer updaten	/user/[:uuid]	PUT
Benutzer holen	/user/[:uuid]	GET
Accountsliste holen	/users	GET

Tabelle 4.1: Auflistung der Endpunkte

count zu löschen, wird der /user/[:uuid] Endpunkt gebraucht, welcher eine DELETE-Methode erwartet.

Damit der Admin das Passwort, E-Mail oder Name anpassen [FA114] kann, wird noch eine weitere PUT-Methode benötigt. Diese solle unter /user/[:uuid] erreichbar sein. Hier soll einfach der veränderte Benutzer mitübergeben und in die Datenbank gespeichert werden. Auch kann hier die Rolle mit verändert werden. Zudem soll mit einer GET-Methode /user/[:uuid] ein bestimmter Benutzer geholt werden.

Wenn alle Accounts angezeigt werden sollen [FA115 / FA1116], wird wie bei FA103 zusätzlich eine Suche und Seitennummerierung benötigt. Außerdem muss bei den Daten unterschieden werden, ob sie aktiviert sind oder nicht, um sie später in verschiedenen Listen anzuzeigen. Sobald der Suchen-Parameter ausgefüllt ist, soll keine Unterscheidung mehr stattfinden. Der Endpunkt soll unter /users erreichbar sein.

4.3 Frontend

Für das Frontend wird die JavaScript-Bibliothek React verwendet. Mit Hilfe von TypeScript Modulsystem, Klassen, Schnittstellen und einem Typsystem kann man

einen schönen, performanten und geordneten Code schreiben. Aber warum wird nicht Angular genutzt? Einer der größten Vorteile ist es, dass nur Funktionen mitgegeben werden, die man auch wirklich benötigt. Im Gegensatz dazu wird bei Angular eine Vielfalt von Funktionen schon mitgegeben. Wenn eine Funktion fehlt, meint Mandler [Man21], wäre es kein Problem, diese nachzuinstallieren, da es eine große Auswahl an flexiblen Bibliotheken gibt. Außerdem ist React leichter hochzuskalieren und mit seiner Komponenten-Architektur effizient zu warten, wodurch NF103 gegeben wäre.

In React denkt man in Komponenten. In der React Dokumentation [Rea21] wird empfohlen, in den Mockups die Komponenten zu markieren. Eine Komponente soll so wie eine Funktion nur eine Sache tun. Wenn eine Komponente zu groß wird, sollte man sie in Unterkomponenten zerlegen.

Zudem muss auf die Architektur der Daten geachtet werden, da man in React den sogenannten „One Way Data Flow“ benutzt. Das bedeutet, dass die Daten unidirektional übergeben werden, also nur von oben nach unten. Das seien Ausnahmen von Events, die vom Nutzer über Aktionen ausgelöst wurden, beschreibt Mandler [Man21]. So entsteht ein einheitlicher Datenfluss.

Um die Implementierung zu erleichtern, wird zu React die UI-Bibliothek Ant Design genutzt. Diese bringt einige Komponenten mit, die eine elegante Benutzeroberfläche unterstützen. Dies verbessert auch die Handhabung und die Benutzbarkeit [NF102].

Bei Erstellung der Mockups wurde der Mobile First-Ansatz verwendet. Hier designt man die Webanwendung zuerst auf kleinen Bildschirmen und dann erst auf den größeren, erklären Schwarz und Frote [Nel20]. Der Vorteil liegt darin, dass man gezwungen wird, minimalistisch zu arbeiten und die Seiten nicht mit Funktionen vollzustellen. Dadurch entsteht automatisch ein geordnetes und übersichtliches Design auf allen Geräten, was die Benutzbarkeit [NF102] wieder enorm verbessert.

In den Mockups wurde sich für zwei Farben entschieden. Ein Rotton, der, auf einem blauen Hintergrund steht. Für eine App-Anmutung sind die oberen Enden abgerundet. Die Abrundungen werden auch immer wieder in der Komponente aufgegriffen. Um den Tipp von der React-Dokumentation umzusetzen, wurden alle Elemente, die später eine Komponente werden, in einem Photoshop-Ordner gesammelt.

In Abbildung 4.1 sieht man die Startseite. Im oberen Bereich sollen die nächsten drei Reservierungen angezeigt werden. Um alle anzeigen zu lassen, soll man auf

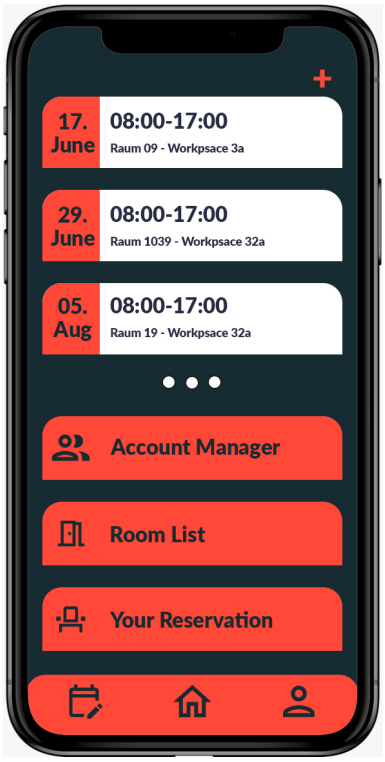

Abbildung 4.1: Startseite Mockup

die drei Punkte klicken können. Um weitere Arbeitsplätze zu reservieren, gibt es das Plus im oberen Bereich oder das linke Symbol in der Navigation. Alle Funktionen sind auch noch mal als Knöpfe im unteren Bereich verfügbar. Der Account Manager-Knopf soll nur angezeigt werden, wenn der Admin eingeloggt ist. In Abbildung 4.2 sieht man die Account-Seite. Hier sollen alle Accounts aufgelistet werden. Im oberen Bereich soll es ein Suchfeld geben und im unteren Bereich soll es die Möglichkeit geben, die Seiten der Accounts zu wechseln. Mit einem Klick auf einen Account soll man auf eine Detail-Ansicht kommen, wo alle Informationen aufgezeigt werden. Wenn man über die Raumliste geht und auf einen Raum klickt, soll man auf die Abbildung 4.3 gelangen. Hier werden alle Details des Raums angezeigt. Unter den Informationen soll ein Raumplan angezeigt werden, in dem man den passenden Arbeitsplatz finden kann. Klickt man auf einen Arbeitsplatz, wird man auf die Detail-Ansicht weitergeleitet, auf der man in einem Kalender seine Reservierung abschließen kann.

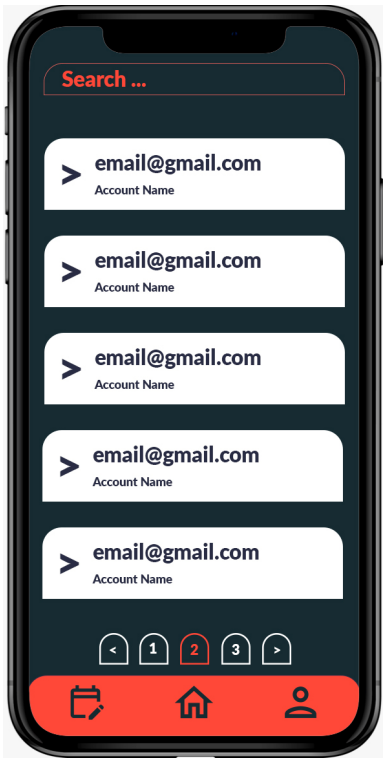

Abbildung 4.2: Account-Seite Mockup

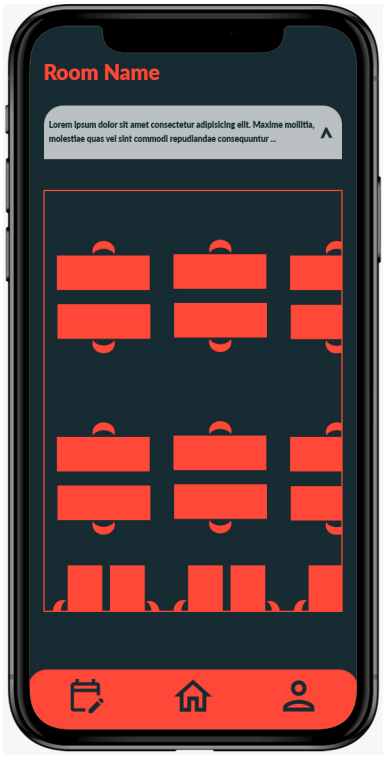

Abbildung 4.3: Raum-Seite Mockup

Kapitel 5

Backend

In diesem Kapitel wird die die Entwicklung des Backend beschrieben. Es wird auf die MongoDB-Datenbank eingegangen und auf die RESTful-API, die mit Go geschrieben wurde.

5.1 Datenbank

Für die Datenbank musste nichts Neues kreiert werden. Es muss lediglich ein MongoDB Test Server erstellt werden, welcher mit Hilfe von Docker erstellt wurde. Die Datenbankstruktur wird vom Go Code erarbeitet. Das ist der sogenannte Code-First-Ansatz. Hierbei konzentriert man sich auf die Domäne der Anwendung und beginnt mit der Erstellung von Klassen oder, im Fall von Go, mit der Struktur und nimmt dabei keine Rücksicht auf die Datenbank.

Alle Strukturen befinden sich im `models`-Ordner. Für den Benutzer werden eine UUID, Name, E-Mail, Passwort und eine Benutzer-Rolle benötigt. E-Mail und Passwort werden später für das Einloggen wichtig sein. Die Benutzer-Rolle ist vom Typ „Role“, welches ein Enum ist, in dem die Rollen abgespeichert sind, zu sehen im Codeblock 5.1.

```
1 (
2     RoleAdmin = "admin"
3     RoleMember = "member"
4     RoleNone  = "none"
5 )
```

Listing 5.1: Role Enum

Das Besondere am Raum ist, dass hier alle Arbeitsplätze und Spezifikationen in Arrays abgespeichert werden. So liegt alles, was mit dem Raum zu tun hat, in einer Struktur, und es kann ganz leicht darauf zugegriffen werden. Am Ende einer Zeile wird erneut definiert, wie man in json oder bson auf die Variablen zugreifen kann. Eine Struktur, wie im Codeblock 5.2, kann einfach in die MongoDB gespeichert werden und erstellt die Felder, wenn sie noch nicht vorhanden sind.

```
1 type Room struct {
2     UUID          string          `json:"uuid" bson:"_id, omitempty"`
3     Name          string          `json:"name,omitempty" bson:"name,omitempty"`
4     Description   string          `json:"description" bson:"description"`
5     Delete        bool            `json:"delete" bson:"delete"`
6     Workspaces    []Workspace     `json:"workspaces" bson:"workspaces"`
7     Specification []Specification `json:"specification" bson:"specification"`
8 }
```

Listing 5.2: Raumstruktur

Abschließend existiert noch eine Reservierungsstruktur. Diese soll eine eigene Collection werden, um schneller auf alle Reservierungen zugreifen zu können. Hier werden neben der Arbeitsplatz UUID und Raum UUID auch die Namen, von Raum und Arbeitsplatz mit abgespeichert. Die Zeiten werden in Unix Timestamps abgespeichert. Im Codeblock 4.1 sieht man auch das Datenbankschema, welches alle Strukturen zeigt.

Um mit der Datenbank zu kommunizieren, muss einmalig eine Verbindung aufgebaut werden. Hierfür kann das eigene MongoDB Go Driver-Paket [Inc17] benutzt werden. Um die Passwörter nicht im Code zu haben sind sie in der .env Datei ausgelagert. Mit `util.GetEnvVariable("MONGODB-USERNAME")`, in Zeile 3 im Codeblock 5.3, lässt sich einfach darauf zugreifen.

```
1  // Set Username and Password
2  credential := options.Credential{
3      Username: util.GetEnvVariable("MONGODB_USERNAME"),
4      Password: util.GetEnvVariable("MONGODB_PASSWORD"),
5  }
6  applyUri := fmt.Sprintf("mongodb://%s:%s/", util.GetEnvVariable("MONGODB_HOST"),
       util.GetEnvVariable("MONGODB_PORT"))
7  // Set client options
8  clientOptions := options.Client().ApplyURI(applyUri).SetAuth(credential)
9
10 Ctx, _ = context.WithTimeout(context.Background(), 10*time.Second)
11
12 // Connect to MongoDB
13 client, err := mongo.Connect(context.TODO(), clientOptions)
```

Listing 5.3: Datenbankverbindung

Mit `mongo.Connect`, in Zeile 13 im Codeblock 5.3, wird versucht, eine Verbindung aufzubauen. Wenn die Verbindung erfolgreich war, wird der Client mit der Datenbank in einer Globalen Variable abgespeichert. Wenn keine Verbindung aufgebaut werden kann, wird das Programm beendet.

5.2 Application Programming Interface

Für die API kann man das Mux-Paket [Mat12] verwenden, das einen Request-Router und einen Dispatcher hat. Damit können eingehende Anfragen zu einem Handler (Funktion) zugeordnet werden. `http.ListenAndServe` startet einen HTTP-Server mit angebender Adresse und einem Handler.

Im Code 5.4 wurde nur ein Port gesetzt [5.4-Z.1], was bedeutet, dass der Server auf localhost:27017 erreichbar ist. Zudem wurden Cross-Origin Resource Sharing Einstellungen verwendet. Dies erhöht die Sicherheit und kann verhindern, dass ein Dritter auf den Server zugreifen kann. „AllowedHeader“ erstellt eine Whitelist [5.4-4]] und erlaubt nur bestimmte Header-Typen-Anfragen zu übermitteln. Auch werden nicht alle Methoden durchgelassen [5.4-Z.5], da bisher der Server nur mit dem Frontend kommunizieren darf. Deshalb wurde die Frontend Server URL auch als einzige Domain gewhitelisted [5.4-Z.7].

```
1 http.ListenAndServe(util.GetEnvVariable("API_PORT"),
2    //Settings Header Cors
3    handlers.CORS(
4       handlers.AllowedHeaders([]string{"X-Requested-With", "Content-Type", "
            Authorization"}),
5       handlers.AllowedMethods([]string{"GET", "POST", "DELETE", "PUT"}),
6       handlers.AllowCredentials(),
7       handlers.AllowedOrigins([]string{util.GetEnvVariable("SERVER_URL")}),
8    )(r))
```

Listing 5.4: HTTP-Server

Damit der Server weiß, auf welche Endpunkte er hören soll, müssen Handler mit den Methoden erstellt werden. Hierfür ist ein Array erstellt worden, in dem alle Endpunkte definiert werden. Dafür wurde die „Route“-Struktur konstruiert, zu sehen im Codeblock 5.5.

```
1 type route struct {
2    Name        string
3    Method      string
4    Pattern     string
5    HandlerFunc http.HandlerFunc
```

```
6     Role        models.Role
7 }
```

Listing 5.5: Route-Struktur

Der Name hat keinen entscheidenden Zweck und dient ausschließlich der Beschreibung für den Quellcode. In der Methode wird die Aufruf-Methode gespeichert, also GET, POST, DELETE oder PUT. Im Pattern wird die Aufrufs-URL, in der HandlerFunc wird die Funktion gesetzt, welche bei dem API-Aufruf ausgeführt werden soll. Zum Schluss wird noch die Rolle gespeichert, mit welcher man auf den Endpunkt zugreifen darf.

Über das Array geht eine Schleife, welche im Codeblock 5.6 zu sehen ist. In dieser werden die Handler-Funktionen der Endpunkte erstellt. Die Funktionen befinden sich alle im selbstgeschriebenen Controller-Paket. Zuerst wird das Pattern gesetzt und dann die Funktion, die ausgeführt werden soll. Die Funktion wird je nach Rolle noch einmal durch eine eigens geschriebene Funktion durchgeschickt. In dieser wird geschaut, ob der Anfrager die Rechte besitzt, die er für den Endpunkt benötigt. Dies wird mit Hilfe des JWT überprüft. Im JWT sind die Benutzerinformationen gespeichert. Für die Middleware ist nur die Rolle relevant. Mit der `GetCurrentUser` Funktion wird der Benutzer anhand der gesetzten Claims im Token geholt. Im weiteren Verlauf wird erläutert, wie sie das macht. Jetzt wird nur noch mit einer Abfrage überprüft, ob er die richtige Rolle für den Endpunkt hat. Falls nicht, bekommt er einen 401-Fehler zurück. Ganz zum Schluss wird noch die Methode definiert.

```
1  for _, route := range getRoutes() {
2     switch route.Role {
3     case models.RoleAdmin:
4        router.Handle(route.Pattern, middleware.Admin(route.HandlerFunc)).Methods(
             route.Method)
5     case models.RoleMember:
6        router.Handle(route.Pattern, middleware.Member(route.HandlerFunc)).Methods(
             route.Method)
7     case models.RoleNone:
8        router.Handle(route.Pattern, route.HandlerFunc).Methods(route.Method)
9     }
10 }
```

Listing 5.6: Route Schleife

5.2.1 Benutzer erstellen

Die Funktion, die aufgerufen wird, ist die `CreateUser`, die im Codeblock 5.7 geschrieben ist. Ihr werden zwei Parameter übergeben, einmal ein `http.ResponseWriter` und der `http.Request`, die gebraucht werden, um mit dem Client zu kommunizieren [5.7-Z.1]. Jede Funktion fängt mit einem `Println` an, damit man in der Konsole immer sehen kann, welche Anfragen reinkommen. Dann wird der Header definiert; da nur JSON-Dateien übersendet werden, wird der `content-type` auf `application/json` gesetzt [5.7-Z.3]. Dann wird versucht, die Daten, die übergeben wurden, in die Benutzer-Datenstruktur zu setzen. Wenn fehlerhafte Daten übergeben wurden, lässt sich der Benutzer nicht erstellen und die Funktion bricht mit einem 400-Fehler ab. Der Fehlercode steht für Bad Request und bedeutet, dass die übergebene Syntax vom Server nicht bearbeitet werden konnte [5.7-Z.8]. Wenn alles stimmt, wird eine Verbindung zur User-Collection aufgebaut. Hierfür wurde eine eigene Hilfefunktion erstellt: `initUserCollection()`. Danach wird die übergebene E-Mail mit der Datenbank abgeglichen, ob diese E-Mail bereits vorhanden ist [5.7-Z.13-18]. Ist das der Fall, bricht die Funktion wieder mit einem 400-Fehler ab. Wenn die E-Mail noch nicht vergeben ist, wird mit dem Passwort weitergearbeitet, welches mit Hilfe des bcrypt Algorithmus gehasht wird [5.7-Z.21]. Das gehashte Passwort wird dann in die Benutzervariable gespeichert und überschreibt das übergebende Passwort. Jetzt bekommt die Variable noch die Rolle None und eine UUID. Dann ist der Benutzer fertig und kann in der Datenbank abgespeichert werden.

```
1  func CreateUser(w http.ResponseWriter, r *http.Request) {
2     fmt.Println("Create User")
3     w.Header().Add("content-type", "application/json")
4     var user models.User
5     //Decode body to user
6     err := json.NewDecoder(r.Body).Decode(&user)
7     if err != nil {
8        http.Error(w, err.Error(), http.StatusBadRequest)
9        return
10    }
11    initUserCollection()
12
13    _, isEmailSet, _ := GetUserByEmail(user.Email)
14    //if user exists with email then UUID length != 0 and return handler
15    if isEmailSet {
16       http.Error(w, "Email already exists", http.StatusBadRequest)
17       return
18    }
19
20    //Hash and Salt the password
```

```
21      hashPassword, _ := bcrypt.GenerateFromPassword([]byte(user.Password), 14)
22      user.Password = string(hashPassword)
23      user.UserRole = models.RoleNone
24      //Create uuid UserID
25      user.UUID = uuid.New().String()
26
27      //Save User in Collection User
28      result, _ := usersCollection.InsertOne(database.Ctx, user)
29      json.NewEncoder(w).Encode(result)
30
31  }
```

Listing 5.7: CreateUser-Funktion

5.2.2 Benutzer aktivieren

Um den Benutzer zu aktivieren, sodass er in der Applikation Eintragungen erstellen kann, gibt es einen Endpunkt für den Admin, den „SetUserRoles“. Der Endpunkt ruft die SetRole Funktion auf, welche sich mit

```
1  params := mux.Vars(r)
2  userUUID, _ := params["uuid"]
```

Listing 5.8: UUID aus URL ziehen

UUID aus der URL holt. Danach holt sie sich die übergebene Rolle aus dem Header und setzt die Benutzerrolle.

```
1  result, _ := usersCollection.UpdateByID(database.Ctx, userUUID, bson.M{"$set": bson
       .M{"role": user.UserRole}})
```

Listing 5.9: Update User-Collection

5.2.3 Benutzer löschen

Der Admin kann auch Benutzer löschen mit dem Endpunkt „DeleteUser“. Die UUID des Users ist wieder in der URL mitgegeben. Anhand dessen wird der User-Eintrag gelöscht.

```
1  result, err2 := usersCollection.DeleteOne(database.Ctx, bson.M{"_id": userUUID})
```

Listing 5.10: Benutzer löschen

Zudem müssen auch alle Reservierungen des Benutzers gelöscht werden. Nur die Reservierungen, die in der Vergangenheit liegen, werden nicht gelöscht, um hier nachvollziehen zu können, wann wo jemand gesessen hat.

```
unix := time.Now().Unix()
  filter := bson.M{"$and": []interface{}{
    bson.M{"start_date": bson.M{"$gte": unix}},
    bson.M{"user_uuid": userUUID}}}
  _, err2 = reservationCollection.DeleteMany(database.Ctx, filter)
```

Listing 5.11: Reservierung vom Benutzer löschen

5.2.4 Alle Benutzer anzeigen

Damit der Admin später eine Liste von allen Usern, verwalten kann, gibt es den „GetUsers“ Endpunkt. Das Besondere hierbei ist die Möglichkeit, die Daten aufzuteilen, sodass nicht alle Benutzer auf einmal angezeigt werden. Es kann dann im Frontend eine Seitennummerierung erstellt werden. Die Seiten und die Größe an Daten, die zurückkommen soll, werden als Parameter an die URL gehängt. Wenn man die erste Seite an Daten haben möchte und 12 Benutzer zurückbekommen möchte, müsste die URL so aussehen `/users?page=0&size=12`. Auch ist eine Suche möglich mit `/users?search=Suche`. Hierbei wird das gesuchte Wort im Namen oder in der E-Mail gesucht. Auch kann mit angegeben werden, ob man aktivierte oder nicht aktivierte Benutzer sucht. Es sei denn man sucht nach einem bestimmten Namen, hier wird immer die ganze Tabelle durchsucht.

```
active := r.URL.Query().Get("active")
pageString := r.URL.Query().Get("page")
sizeString := r.URL.Query().Get("size")
search := r.URL.Query().Get("search")
```

Listing 5.12: URL-Parameter: Alle Benutzer anzeigen

Die Funktion, die beim Aufruf ausgeführt wird, selektiert zuerst die Parameter. Danach wird ausgerechnet, wie viele Daten bei der Datenbankabfrage ausgelassen werden sollen, sodass man einzelne Daten aufrufen kann. Die erste Seite mit Daten wird angefragt, also wird 0*12 gerechnet und es werden die ersten 12 Benutzer zurückgegeben. Wenn jetzt die zweite Seite angefragt wird, ist die Rechnung 1*12, die ersten 12 Daten werden übersprungen und die nächsten 12 Benutzer werden angezeigt. Dies bedeutet aber auch, dass man bei der Datengröße bleiben muss.

Demnach kann man nicht die erste Seite mit 12 Benutzer zurückbekommen und auf der zweiten dann nur 8 erhalten. Das würde zu Problemen führen.

```
result, err := usersCollection.Find(database.Ctx, bson.M{},
   options.Find().SetProjection(bson.M{"password": 0}).SetLimit(int64(size)).
       SetSkip(skip))
```

Listing 5.13: Datenbank abfrage alle Benutzer anzeigen

Zudem werden nicht nur die Benutzer zurückgegeben, sondern auch die Information von den Seiten, welche Seite zurückgeben wurde, wie viele Seiten es insgesamt gibt und wie viele Element in der Datenbank vorhanden sind. Die Seitenzahlen werden ausgerechnet. Hierfür wird die Collection gezählt, um herauszufinden, wie viele Benutzer sich in ihr befinden.

```
collectionLength, _ := usersCollection.CountDocuments(database.Ctx, bson.M{})
```

Listing 5.14: Größe der Collection

Um die Total-Seiten zu ermitteln, wird die Collection-Länge durch die Datengröße gerechnet und der Wert immer aufgerundet. Da es keine Funktion dafür gibt, wird dies mit einer Abfrage geregelt. Es wird geprüft, ob der Float-Wert größer ist als der Integer-Wert [5.15-Z.4]. Wenn dies der Fall ist, wird die Seitenanzahl der Integer-Wert plus eins gerechnet. Wenn das nicht der Fall ist, können die Werte nur gleich groß sein und die Seitenzahl ist der Wert. Damit sind die Daten auf die Seiten perfekt aufgeteilt [5.15-Z.5-8].

```
stepsFloat := float64(collectionLength) / float64(size)
stepsInt := collectionLength / int64(size)
var steps int64
if stepsFloat > float64(stepsInt) {
   steps = stepsInt + 1
} else {
   steps = stepsInt
}
```

Listing 5.15: Berechnungen für Seitennummerierung

Wenn der Suchparameter ausgefüllt ist, werden alle Filter und Größen oder Seiten ignoriert. Es wird ein Regex mit dem Suchparameter erstellt [5.16-Z.1]. Nach dem Regex werden die Namen und E-Mails in der Datenbank durchsucht.

```
regex := bson.M{"$regex": primitive.Regex{Pattern: search, Options: "i"}}
filter = bson.M{"$or": []interface{}{
   bson.M{"name": regex},
   bson.M{"email": regex},
}}
```

Listing 5.16: Suchfilter

5.2.5 Login

Wenn der Benutzer Member oder Admin ist, kann er sich einloggen. Hierfür wird der Endpunkt Login erstellt. Übermittelt werden sollen die Login-Daten, die aus E-Mail und Passwort bestehen. Zuerst wird erneut geprüft, ob der Benutzer mit der E-Mail in der Datenbank vorhanden ist. Ist das der Fall, wird überprüft, ob das richtige Passwort zum Benutzer übergeben wurde.

```
1 if err := bcrypt.CompareHashAndPassword([]byte(userInDatabase.Password), []byte(
      login.Password)); err != nil {
2    handler.HttpErrorResponse(w, http.StatusBadRequest, "User or Password wrong")
3    return
4 }
```

Listing 5.17: Passwortüberprüfung

Wenn das Passwort, mit demjenigen in der Datenbank übereinstimmt, wird ein JWT erstellt. Hierfür wurde die Hilfsfunktion `CreateToken` erstellt, die den Benutzer mit übergeben bekommt. Der Token soll nicht nur der Authentifizierung dienen, sondern es sollen sich auch die wichtigsten Daten vom Benutzer in dem Token befinden [5.18-Z.4-8]. Deswegen werden UUID, Name, Rolle und die E-Mail mit im Token gespeichert. So können später viele Datenbankabfragen vermieden werden. Außerdem wird eine Ablaufzeit hinzugefügt, sodass der Token nur für zwei Stunden nutzbar ist [5.18-Z.9]. Da der JWT nur im Backend benötigt wird, wird ein symmetrischer Algorithmus für die Verschlüsselung genutzt, HS256. Der private Schlüssel liegt auch in der `.env`-Datei.

```
1  func CreateToken(user models.User) (string, error) {
2     var err error
3     //Creating Access Token
4     atClaims := jwt.MapClaims{}
5     atClaims["role"] = user.UserRole
6     atClaims["uuid"] = user.UUID
7     atClaims["name"] = user.Name
8     atClaims["email"] = user.Email
9     atClaims["exp"] = time.Now().Add(time.Hour * 2)
10    at := jwt.NewWithClaims(jwt.SigningMethodHS256, atClaims)
11    token, err := at.SignedString([]byte(util.GetEnvVariable("SECRETKEY")))
12    if err != nil {
13       return "", err
14    }
```

```
15     return token, nil
16 }
```

Listing 5.18: CreateToken Funktion

Nun kann der Token an den Client übermittelt werden. Dafür wird mit Hilfe des HTTP-Paket ein Cookie erstellt. Dieser bekommt ebenfalls eine Ablaufzeit von 2 Stunden und ist HttpOnly, so kann kein Script auf diesen Cookie zugreifen. Dies ist auch nicht notwendig, weil der Cookie nur im Backend benötigt und bei jeder Anfrage im Header mit übermittelt wird. Dies ist die einfachste Lösung, um eine Authentisierung zu erstellen. Hierbei muss beachtet werden, dass der Token bei einer nicht verschlüsselten Anfrage (http) einfach ausgelesen werden kann. So könnte zum Beispiel ein „Man-in-the-Middle“-Angreifer den Token recht leicht stehlen und für seinen Browser verwenden. Dieses Problem könnte man mit einem aufwändigen Public-/Private-Key-Verfahren beheben. Auch wenn man die Anfragen verschlüsselt, sollte man verhindern, dass der Token ohne weiteres für Anfragen verwenden kann.

```
1 cookie := http.Cookie{
2    Name:    "jwt",
3    Value:   token,
4    Path:    "/",
5    Expires: time.Now().Add(time.Hour * 2),
6    HttpOnly: true,
7 }
8 http.SetCookie(w, &cookie)
```

Listing 5.19: Cookie erstellen

Zum Schluss wird der Benutzer noch als JSON ohne Passwort an den Client übermittelt.

```
1 userInDatabase.Password = ""
2 json.NewEncoder(w).Encode(userInDatabase)
```

Listing 5.20: Übermittlung von Daten

5.2.6 Logout

Die Logout-Funktion, die aufgerufen wird, erstellt einen neuen Cookie ohne JWT und setzt die Ablaufzeit in die Vergangenheit. So wird der Cookie gelöscht.

```
1 cookie := http.Cookie{
2    Name:    "jwt",
```

```
3     Value:   "",
4     Path:    "/",
5     Expires: time.Now().Add(-time.Hour * 2),
6     HttpOnly: true,
7  }
```

Listing 5.21: Cookie löschen

5.2.7 Nutzer anhand des Tokens bekommen

Der `GetUserByToken`-Endpunkt ist wichtig, damit nicht immer der Benutzer aus der Datenbank gesucht werden muss. Die `GetUserByToken`-Funktion wird ausgeführt und in dieser wird mit Hilfe der `GetCurrentUser` der Benutzer aus dem Token geholt. Im Codeblock 5.7 ist die Funktion zu sehen. Die `GetCurrentUser`-Funktion bekommt den http Request mit übergeben und gibt einen Benutzer und ein Error zurück [5.4-Z.1]. Aus dem Request holt sie sich den JWT Cookie, der beim Login gesetzt wurde. Wenn der Cookie nicht gesetzt ist, bricht die Funktion ab und gibt einen leeren Benutzer mit einem Error zurück. Ist der Cookie vorhanden, wird überprüft, ob der Token valide ist und wird wieder decoded [5.4-Z.5-10]. So lässt sich erneut auf die Token Claims zugreifen, auf denen die Benutzerdaten gespeichert worden sind [5.4-Z.12-20]. Die Daten werden dann in eine Benutzerstruktur gepackt und ausgegeben.

```
1  func GetCurrentUser(r *http.Request) (models.User, error) {
2     cookie, err := r.Cookie("jwt")
3     var user models.User
4     if err != nil { }
5     token, err := jwt.Parse(cookie.Value, func(token *jwt.Token) (interface{}, error
          ) {
6        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
7           return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"
                ])
8        }
9        return []byte(util.GetEnvVariable("SECRETKEY")), nil
10    })
11    if err != nil { }
12    claims, ok := token.Claims.(jwt.MapClaims)
13    if ok && token.Valid {
14       user = models.User{
15          UUID:     claims["uuid"].(string),
16          Name:     claims["name"].(string),
17          UserRole: models.Role(claims["role"].(string)),
18          Email:    claims["email"].(string),
19       }
20
21       return user, nil
```

```
22      }
23      return user, nil
24  }
```

Listing 5.22: GetCurrentUser

5.2.8 Reservierung erstellen

Um eine Reservierung zu erstellen, muss eine Reservierungsstruktur übermittelt werden. Wenn alles richtig übergeben wurde, wird zuerst überprüft, ob der Zeitraum in der Zukunft und nicht in der Vergangenheit liegt.

```
1  if time.Now().Unix() > reservation.StartDate {
2      handler.HttpErrorResponse(w, http.StatusBadRequest, "Time lies in the past")
3      return
4  }
```

Listing 5.23: Überprüfung der Zeit 1

Danach wird überprüft, ob das Startdatum auch vor dem Enddatum liegt, damit keine fehlerhafte Reservierung erstellt werden kann.

```
1  if reservation.EndDate < reservation.StartDate {
2      handler.HttpErrorResponse(w, http.StatusBadRequest, "End time is before the
           start time")
3      return
4  }
```

Listing 5.24: Überprüfung der Zeit 2

Zusätzlich wird überprüft, ob der Arbeitsplatz zum angefragten Datum frei ist. Hierfür gibt es die Helferfunktion `FindReservationBetweenTime`, diese bekommt die Arbeitsplatz-UUID, Startdatum und Enddatum übergeben. In der Funktion wird eine Datenbankabfrage erstellt, die prüft, ob in dem Zeitraum schon eine Reservierung vorhanden ist.

```
1  bson.M{"$or": []interface{}{
2      bson.M{"$and": []interface{}{
3          bson.M{"start_date": bson.M{"$gte": start}},
4          bson.M{"start_date": bson.M{"$lte": end}},
5      }},
6      bson.M{"$and": []interface{}{
7          bson.M{"end_date": bson.M{"$lte": start}},
8          bson.M{"end_date": bson.M{"$gte": end}},
9      }},
10     bson.M{"$and": []interface{}{
11         bson.M{"start_date": bson.M{"$lte": start}},
```

```
12          bson.M{"end_date": bson.M{"$gte": end}},
13      }},
14  }}
```

Listing 5.25: Überprüfung der Zeit 2

Ein Filter wird erstellt. Der Einfachheit halber werden die Daten in der Datenbank mit DB gegengezeichnet. Eine Reservierung besteht in dem Zeitraum, wenn das DB-Startdatum größer gleich des Startdatums ist und das DB-Startdatum kleiner gleich des Enddatums ist. Gleiches gilt aber auch in dem Zeitraum, wenn das DB-Enddatum kleiner gleich des Startdatums ist und das DB-Enddatum größer gleich des Enddatums ist. Als letzte Möglichkeit ergibt sich, dass das DB-Startdatum kleiner gleich dem Startdatum und das DB-Enddatum größer gleich dem Enddatum ist, was im Codeblock 5.25 in der bson-Schreibweise zu sehen ist.

Wenn nach der Abfrage eine Reservierung zurückkommt, dann ist der Platz schon belegt. Wenn nichts zurückkommt, kann die Reservierung angelegt werden. Der aktuelle Benutzer wird geholt und die UUID wird zur Struktur hinzugefügt. Zusätzlich werden die Namen von Raum und Arbeitsplatz aus der Datenbank geholt. Auch diese werden in der Reservierungsstruktur gespeichert.

5.2.9 Reservierung erstellen

Da die Namen in der Reservierung gespeichert werden, können die Namen von den Arbeitsplätzen nicht abgeändert werden. Nur die Beschreibung und die Position sollen abgeändert werden. Aus der URL wird die Raum UUID genommen und anhand derer der Raum aus der Datenbank geholt. Danach wird aus der Anfrage der übergebene Raum abgespeichert. Nun wird durch die übergebenen Arbeitsplätze im Raum iteriert. Alle Arbeitsplätze, die keine UUID haben sind neue Plätze und bekommen eine UUID zugeteilt. Nun muss überprüft werden, ob Arbeitsplätze gelöscht wurden. Hierfür wird eine Schleife durch die alten Arbeitsplätze durchgeführt. In dieser Schleife wird noch mal eine Schleife über die neuen Elemente ausgeführt. Nun wird überprüft, ob die UUID in dem neuen Element enthalten ist. Ist das nicht der Fall, ist der Arbeitsplatz gelöscht worden und die UUID wird in ein Array gespeichert. Nach dem die Schleife durchgelaufen ist, liegt ein Array mit allen Arbeitsplatz-UUIDs vor, die gelöscht wurden. Da es eine Schleife in einer Schleife ist, ist die Komplexität $O(n^2)$, was ziemlich hoch ist. Hier könnte man im weiteren Verlauf einen passenderen Algorithmus anwenden, um die Suche zu verbessern. Aber da

dieser Endpunkt nur für den Admin wichtig und für den User irrelevant ist, ist das erst einmal zu vernachlässigen.

```
for _, oldElement := range room.Workspaces {
   isNotInNew := true
   for _, newElement := range newRoom.Workspaces {
      if newElement.UUID == oldElement.UUID {
         isNotInNew = false
      }
   }
   if isNotInNew {
      deleteList = append(deleteList, oldElement.UUID)
   }
}
```

Listing 5.26: Arbeitsplatz gelöschte Elemente

Nun wird, wenn der Array nicht leer ist, die UUID in der Reservierung-Collection gesucht und alle passenden Reservierungen gelöscht, damit keine Reservierung ohne Arbeitsplatz existiert.

```
_, err := reservationCollection.DeleteMany(database.Ctx, bson.M{"workspace_uuid":
    bson.M{"$in": deleteList}})
```

Listing 5.27: Reservierung löschen mit gelöschtem Arbeitsplatz

5.2.10 Update User

Die Account-Daten soll erstmal nur der Admin verwalten können, da man für sichere Veränderungen von Daten, eine doppelte Authentifizierung bräuchte, welche noch nicht entwickelt wurde. Der Endpunkt ist unter /user/:uuid erreichbar und erwartet eine PUT-Methode. Übergeben wird ein Nutzer, das Passwort darf fehlen, nicht aber Name, E-Mail und Rolle. Diese Werte werden immer mit in die Update-Methode reingeschrieben. Zuerst wird der User anhand der UUID in der URL gesucht. Dann wird geschaut, ob sich die E-Mail verändert hat. Ist das der Fall wird geschaut, ob die neue E-Mail schon vergeben ist. Wenn ein User mit der E-Mail-Adresse gefunden wurde, wird ein Error ausgegeben.

Wenn das Passwort gesetzt ist, muss es erst wieder gehasht werden. Dann wird die Update Methode erstellt. Hierbei wird noch überprüft, ob eine valide Rolle hinzugefügt wurde. Dannach wird der User mit den neuen Daten ersetzt und zurückgegeben.

```
update = bson.D{{"$set", bson.M{"name": user.Name, "password": user.Password, "
    email": user.Email, "role": user.UserRole}}}

```

```
3 result, err := usersCollection.UpdateByID(database.Ctx, userUUID, update)
```

Listing 5.28: Update User

Kapitel 6

Frontend

In diesem Kapitel geht es um das Frontend. Hier wird beschrieben, wie die React-Applikation entwickelt wurde und wie sie mit dem Backend kommuniziert.

6.1 Package.json

In der package.json befinden sich alle Abhängigkeiten des Projekts. Hier werden der Name des Projektes und alle Pakete, die gebraucht werden, definiert. Bei den Paketen unterscheidet man zwischen dependencies und devDependencies. Die Pakete, die für die Applikation nur bei der lokalen Entwicklung benötigt werden, sind devDependencies-Pakete. In dependencies werden die Pakete reingelegt, welche in der Produktion benötigt werden.

Der wichtigste Punkt für die Entwicklung ist, dass alles schnell getestet werden kann und dass der Quellcode korrekt ist. Damit die Applikation lokal laufen kann, gibt es den Parcel-Bundler. Dieses Paket kümmert sich darum, dass im Browser alles erreichbar ist. Zudem komprimiert er den Quellcode, sodass die Applikation Dateigrößen verringert und später schneller wieder aufgerufen werden kann. Außerdem ist ein Linter installiert, der die Quellcodequalität überprüft. Der Linter überprüft, dass keine überflüssigen Variablen oder Paketimporte vorhanden sind und dass die Einrückungen vom Quellcode stimmen. Infolgedessen entsteht automatisch ein übersichtlicher Quellcode.

Zu den dependencies-Paketen gehören zum Beispiel antd oder auch react. Weiterd Pakete werden im laufenden Kapitel noch erwähnt.

```
1 "dependencies": {
```

```
2   "antd": "^4.9.3",
3   "react": "^17.0.2",
4 }
```

Listing 6.1: Dependencies package.json

6.2 Datenstruktur

Im Hauptordner befinden sich alle setup-Dateien wie beispielsweise die `package-.json` oder auch die `.eslintrc.yml`, welche die Einstellung des Linter beinhalten. Die Hauptdateien befinden sich im `public`-Ordner, in diesem liegt noch die `index.html`. Hier wird in die Sektion mit der Id „`client`“ die ganze Applikation hereingeladen. Darum kümmert sich die `index.tsx`, auf die später auch noch eingegangen werden wird.

```
1 <!doctype html>
2 <html lang="en">
3   <head>  </head>
4   <body>
5     <section id="client"></section>
6     <script src="../src/index.tsx"></script>
7
8   </body>
9 </html>
```

Listing 6.2: Index.html

Die Unterordner `.cache`, `dist` und `node-modules` werden automatisch generiert. In den `node-modules` Ordner werden alle Pakete abgespeichert.

Der wichtigste Ordner ist der `src`-Ordner, hier befindet sich der gesamte Quellcode der Applikation. Hier befinden sich der `api`- und der `client`-Ordner. Im `client`-Ordner ist der Quellcode für die Seiten, Komponenten und das, Styling, also alles, was der Nutzer später sieht, vorhanden. Im `api`-Ordner sind alle Anfragen an das Backend abgelegt. In diesem befindet sich ein `config.ts` und darüber hinaus für jede Anfrage ein passender Ordner, in dem die Anfrage und die Dateninterfaces liegen. Im `client`-Ordner sind der `assets`-, `componets`-, `context`-, `pages`- und `routes`-Ordner positioniert. So sind alle Funktionen schön geordnet und können schnell gefunden werden.

6.3 Backendverbindung

Die API-Calls werden mit Hilfe der axios- und react-query Bibliothek getätigt. Durch Axios werden die API-Calls übersichtlich. Ein Beispiel:

```
1 export const getCurrentUser = () => api.get<User>('/user').then(res => res.data).
      catch(ErrorHandler)
2
3 export const register = (register: Register) => api.post<User>('user/', register).
      then(res => res.data).catch(ErrorHandler)
4
5 export const login = (login: Login) => api.post<User>('user/login', login).then(res
       => res.data).catch(ErrorHandler)
6
7 export const logout = () => api.post<User>('user/logout', {}).catch(ErrorHandler)
```

Listing 6.3: API-Calls

Vier Zeilen Code, vier Anfragen. Die api-Variable wird einmal mit ihren Einstellungen im Projekt definiert und dann lassen sich alle http-Methoden erstellen. In der Applikation werden die Standardeinstellung von Axios benutzt, nur die baseUrl und Credentials-Einstellungen wurden angepasst. Die Credentials werden standardgemäß nicht mit übergeben. Da bei der Authentifizierung Cookies benötigt werden, wird dies abgeändert. Zudem wird bei einem Fehler der ErrorHandler ausgeführt, welcher dem Nutzer den Fehler deutlich anzeigt oder auf bestimmte Fehler-Codes wie zum Beispiel bei einem 403 reagiert. Bei dem Fehler-Code wird der Nutzer direkt auf die Login-Seite weitergeleitet. Damit alle Daten synchronisiert, aktualisiert und zwischengespeichert werden, wird React Query benutzt. Die Bibliothek fügt zwei Hooks, useQuery und useMutation, hinzu.

Die useQuery Funktion bekommt einen beliebigen Key und eine asynchrone Funktion mit übergeben. Zurück gibt sie einige Werte, die den Status angeben, wie zum Beispiel `isLoading`. Die Funktion wird beim Aufruf der Seite oder Komponente automatisch ausgeführt und löst die asynchrone Funktion aus, welche Daten zurückgegeben und welche in der `data`-Variable gespeichert werden. So wird zum Beispiel abgefragt, ob der Benutzer eigenloggt ist und ob er einen validen Token besitzt.

```
1 const { data: currentUser, isLoading } = useQuery('currentUser', getCurrentUser, {
2   retry: false,
3   enabled: cookies.login === 'true',
4   onError: () => { history.push('/login') }
5 })
```

Listing 6.4: useQuery

Zudem werden noch Optionen hinzugefügt, dass die Anfrage nur einmal abgesendet wird und auch nur, wenn der Login-Cookie gesetzt ist.

Der Login-Cookie wird beim Einloggen zusätzlich mit dem JWT-Cookie vom TypeScript erstellt. Diese Funktion dient dazu, dass das Frontend informiert wird, wann der JWT-Cookie abläuft. Da es sich um einen http-Only-Cookie handelt, kann TypeScript nicht auf den Cookie zugreifen. Deswegen wird ein identischer Cookie vom Frontend ohne JWT und nur mit einem Booleschen Wert erstellt. Wenn also der Login-Cookie abgelaufen ist, ist es auch der JWT-Cookie.

Wenn hier ein Error zurückkommt, wird man zurück auf die Login-Seite geschickt [6.4-Z.4]. Durch das Setzen des Keys „currentUser“ wird der Benutzer gecached [6.4-Z.1]. So wird nur im Hintergrund der Benutzer synchronisiert und aktualisiert. Dadurch muss der Nutzer nicht immer auf die Rückmeldung des Backends warten. So verringert sich die Ladezeit von Daten.

Die `useMutation`-Funktion wird für Erstellen/Aktualisieren/Löschen verwendet. Wie useQuery wird eine asynchrone Funktion mitgegeben, zudem werden die Daten wieder in der data-Variable gespeichert. Zudem wird eine Mutationsfunktion zurückgegeben, welche die useMutation triggert. So wird zum Beispiel der Benutzer aktualisiert. Der updateUser-Aufruf wird in die Mutation gesetzt.

```
1  const { mutate: userUpdate } = useMutation(updateUser, {
2   onMutate: () => {
3     queryClient.setQueryData(‘account-${match.params.uuid}‘, data => {
4       return data
5     })
6   },
7   onSuccess: () => {
8     message.success(’Update’)
9     setEdit(false)
10  }
11 })
```

Listing 6.5: useMutation

Jetzt kann man bei einem Klick-Event zum Beispiel die mutate-Funktion ausführen, welche dann die übergebenen Daten an den udpdateUser-Aufruf weitergibt.

```
1  userUpdate({ uuid: data.uuid, name, email, role, password })
```

Listing 6.6: useMutation update User

6.4 Benutzer im Context

Der eingeloggte Benutzer wird auf vielen Seiten in der Applikation gebraucht, um seine Reservierung zu bekommen oder seine Rolle abzufragen. In React werden Daten nur von oben nach unten weitergegeben. Damit nicht jede Komponente den Benutzer übergeben muss, gibt es den useContext Hook von React. Mit diesem Hook wird die Möglichkeit geboten, den Benutzer zwischen den Komponenten zu verteilen, ohne dass ein Request explizit durch jede Ebene geleitet werden muss.

Damit der Benutzer im Context gespeichert wird, muss eine Komponente erstellt werden. Diese holt sich den Benutzer aus dem Backend und fügt ihn als Variable hinzu.

```
1 return <UserContext.Provider value={currentUser}>
2     {children}
3   </UserContext.Provider>
```

Listing 6.7: Benutzer Context Provider

Das children-Element wird benötigt, damit die Komponente um die Applikation gewrappt werden kann.

```
1 function App () {
2   return (
3       <Router>
4         <UserProvider>
5             <Paths/>
6         </UserProvider>
7       </Router>
8   )
9 }
```

Listing 6.8: App-Komponente

Nun kann man mit useContext (UserContext) überall in der Applikation auf den Benutzer zugreifen.

6.5 URL-Baum

Der URL-Baum wird in der `Paths.tsx` erstellt und in der `App.tsx` hinzugefügt. Wenn man nicht eingeloggt ist, besteht lediglich die Möglichkeit, auf die Login- und Registrierungsseite zu gehen. Wenn man versucht, auf eine weitere Seite zu gehen, wird man automatisch wieder auf die Login-Seite geleitet [6.9-Z.5]. Für den eingeloggten Benutzer gibt es die Startseite `/`, eine Profil-Seite `/user`, die

Seite	Pfad	Rolle
Login	/login	none
Registrien	/register	none
Startseite	/	member
Benutzer	/user	member
Räume Liste	/rooms	member
Raum Detail	/room/:uuid	member
Benutzer Reservierungen	/user/reservation/	member
Arbeitsplatz Reservierungen	/workspace/:room/:workspace	member
Account Liste	/accounts	admin
Account Details	/account/:uuid	admin

Tabelle 6.1: Übersicht Seiten

Raumliste `/rooms`, die Raum-Detail-Seiten `/room/:uuid`, die Arbeitsplatz-Detail-Seite `/workspace/:room/:workspace`, die Reservierungs-Seite für den Benutzer `/user/reservation`, darüber hinaus noch die Accounts-Seite und die Account-Detail-Seite, die nur für den Admin erreichbar sind `/accounts`. Damit die URL-Verteilung funktioniert, wird die React-Router-Dom Bibliothek genutzt. Diese funktioniert wie ein Switch-Case. Sobald die URL zu einem Path passt, wird die angegebene Seite geladen. Aber es werden nur die Paths geladen, die der Benutzer benutzen darf. Im Codeblock 6.9 Zeile 1 und 18 wird überprüft, ob der Benutzer die passende Rolle hat.

```
1  function Paths () {
2    const currentUser = useCurrentUser()
3    if (!currentUser || currentUser.role === "none") {
4      return <Switch>
5        <Route exact path="/login" component={LoginPage}/>
6        <Route exact path="/register" component={RegisterPage} />
7        <Redirect to="/login" />
8      </Switch>
9    } else {
10     return <>
11       <MenuBar/>
12       <Container>
13         <Switch>
14           <Route exact path="/" component={Home} />
15           <Route exact path="/user" component={Profile} />
16           <Route exact path="/rooms" component={RoomList} />
17           <Route path="/room/:uuid" component={RoomDetail} />
18           <Route path="/user/reservation/" component={ReservationList} />
19           <Route path="/workspace/:room/:workspace" component={WorkspaceDetail} />
20           { currentUser.role === "admin" && <>
21               <Route exact path="/accounts" component={AccountList} />
22               <Route exact path="/account/:uuid" component={AccountDetail} />
23             </>
```

```
24          }
25          <Redirect to="/" />
26        </Switch>
27      </Container>
28    </>
29  }
30 }
```

Listing 6.9: URL-Baum

6.6 Menü

Damit das Menü auf jeder Seite angezeigt wird, wird es in der obersten Ebene, in der Paths.tsx. eingebunden. Damit die Menü-Komponente zwischen Desktop - und Mobilegröße unterscheiden kann, da auf dem Desktop eine Sidebar und auf der mobilen eine App-Leiste gebraucht wird, wird ein selbst erstellter Hook gebaut. Der useMobile Hook gibt nur ein Boolean zurück. Dieser überprüft, ob das Browser-Fenster kleiner als 940 Pixel breit ist. Wenn das der Fall ist, soll eine mobile Ansicht der Applikation angezeigt werden und deswegen gibt der Hook den Wert true zurück. Sobald das Fenster sich neu skaliert, wird der Hook ausgelöst.

```
1  const [isMobile, setIsMobile] = useState(window.innerWidth < 940)
2 useLayoutEffect(() => {
3   function updateMobile () {
4     setIsMobile(window.innerWidth < 940)
5   }
6   window.addEventListener('resize', updateMobile)
7 })
8 return isMobile
```

Listing 6.10: useMobile Hook

Damit der Menüpunkt hervorgehoben wird, der gerade offen ist, wird der useEffect Hook benutzt. Dieser Hook wird immer ausgeführt, wenn der React DOM aktualisiert wird. In diesem wird mit der `match`-Funktion überprüft, welcher Path aktiviert ist. Match wird benutzt, da auch die Menüpunkte hervorgehoben werden sollen, wenn man sich auf einer Unterseite befindet.

```
1 useEffect(() => {
2   const path = history.location.pathname
3   if (path === '/') setCurrentPage('/')
4   if (path.match(/\/room/)) setCurrentPage('/rooms')
5   if (path.match(/\/workspace/)) setCurrentPage('/rooms')
6   if (path.match(/\/user/)) setCurrentPage('/user')
7   if (path.match(/\/user\/reservation/)) setCurrentPage('/user/reservation')
```

```
8   if (path.match(/\/accounts/)) setCurrentPage('/accounts')
9 })
```

Listing 6.11: URL Matches

Mit der `isMobile`-Variable wird geprüft, welches Menü gerendert werden soll. Mit der Zeile 4 im `className` 6.21 wird die CSS-Klasse gesetzt, wenn man sich auf der Seite befindet.

```
1  if (isMobile) {
2    return <div className="mobile-menu">
3           <div className="menu-icons">
4               <div onClick={() => changePage('/rooms')} className={`flex-center ${
                    currentPage === '/rooms' ? 'active' : ''}`}>
5                   <CalendarTodayOutlined />
6               </div>
7
8           </div>
9       </div>
10 } else {
11   return <div className="menu" >
12            <div onClick={() => changePage('/')} className={`${currentPage === '/' ?
                  'active' : ''}`}>
13              <HomeOutlined /> Home
14            </div>
15
16            <Button onClick={() => mutate()}>Logout</Button>
17   </div>
18 }
```

Listing 6.12: App-Bar und Menü

Im Desktop-Menü gibt es noch einen Logout-Button. Auch hier wird wieder React-Query verwendet. Beim Klick auf den Button wird eine API-Anfrage gesendet. Wenn diese erfolgreich war, wird der Login-Cookie gelöscht [6.13-Z.2-3], der Benutzer wird aus dem Cache gelöscht und man wird auf die Login-Seite weitergeleitet [6.13-Z.5].

```
1  const { mutate } = useMutation(logout, {
2   onSuccess: () => {
3     removeCookies('login')
4     queryClient.setQueryData('currentUser', undefined)
5     location.href = '/login'
6   }
7 })
```

Listing 6.13: Logout-Funktion

6.7 Kalender

Die Kalender waren eine besonders große Herausforderung, da der Kalender alle Daten eines Monats darstellen muss. Schnell wurde klar, dass ein eigener Event-Kalender zu aufwändig geworden wäre. Also musste eine passende Bibliothek gefunden werden. Viele Bibliotheken waren nicht mehr mit der 17 Version von React kompatibel, sodass es nicht einfach war, eine solche zu finden. Die eventcalendar-Bibliothek [mob] wäre die perfekte Wahl gewesen. Diese orientiert sich stark an den bekannten Kalendern wie denen von Apple, Android oder Windows. Die Bibliothek hätte alle wichtigen Funktionen wie etwa Responsiveness schon von Haus aus dabei. Der Preis für das günstigste Modell liegt aber bereits bei 235 Euro. Also musste weiter geschaut werden. Weitere Bibliotheken waren sehr flexibel im Styling, aber stellten nicht wirklich Event-Kalender dar. Bei diesen hätte man erst per Klick auf einzelne Tage die Reservierungen erhalten. Gleichzeitig bedeutet jeder Klick darauf einen API-Aufruf. Abschließend fiel die Entscheidung auf den Fullcalendar-Bibliothek [Ful21] in der Desktop-Ansicht. Für die Mobile-Ansicht wurde die revo-calender-Bibliothek [Mol20] genutzt. Beide Ansichten werden in den nächsten Abschnitten weiter beschrieben.

6.7.1 Kalender Desktop-Ansicht

Auf der Startseite wird auf der Desktop-Ansicht ein Kalender mit allen getätigten Reservierungen angezeigt. Dies ist eine Komponente, die auch noch für die Reservierung wichtig wird. Auf der Startseite lassen sich die Reservierungen nicht bearbeiten. Man kann sie nur anklicken und wird zum passenden Arbeitsplatz weitergeleitet. Auch hier gibt es dieselbe Komponente, nur lassen sich die Reservierungen zusätzlich erstellen und löschen.

Der Kalender, der in der Desktop-Version gezeigt wird, wurde mit Hilfe der Fullcalendar-Bibliothek [Ful21] erstellt. Der Kalender hat viele Parameter, die mit übergeben werden können. Der Codeblock 6.14 beginnt mit der Höhe des Kalenders, welche fest auf 700px gesetzt ist [6.14-Z.2]. Der `plugins`-Parameter dient weiteren Ansichten, die man sich anzeigen lassen kann [6.14-Z.3]. So kann man nicht nur in einer Monatsansicht die Termine anschauen, sondern auch in einer Wochen- oder Tagesansicht. In der header-Toolbar lässt sich die Navigation des Kalenders

setzen [6.14-Z.4-8]. Damit man aber überhaupt Termine sieht und bearbeiten kann, gibt es die Parameter `datesSet`, `eventClick` und `select` [6.14-Z.14-17].

```
<FullCalendar
    height={700}
    plugins={[dayGridPlugin, timeGridPlugin, interactionPlugin]}
    headerToolbar={{
      left: 'prev,next today',
      center: 'title',
      right: 'dayGridMonth,timeGridWeek,timeGridDay'
    }}
    initialView='dayGridMonth'
    selectable={true}
    selectMirror={true}
    dayMaxEvents={true}
    events={currentEvents}
    select={handleDateSelect}
    eventContent={renderEventContent} // custom render function
    eventClick={handleEventClick}
    datesSet={getEvents}
/>
```

Listing 6.14: Full Calendar

`DatesSet` wird aufgerufen, wenn der Kalender gerendert oder der Zeitraum verändert wird. Zurückgegeben wird ein `DatesSetArg`. In diesem sind Startdatum und Enddatum des Kalanders gespeichert. Die Werte werden in einen Unix Timestamp umgewandelt und an das Backend gesendet. Wenn der Kalender-Komponente Raum-UUID und Arbeitsplatz-UUID übergeben werden, wird nach den Reservierungen eines Arbeitsplatzes gesucht. Wenn nur ein Benutzer mitgegeben wird, wird nach allen Reservierungen des Benutzers gesucht [6.15-Z.3-4].

```
const getEvents = async (calendarZone: DatesSetArg) => {
 let data: Reservation[]
 if (workspace && roomUuid) data = await updateReservation({ workspaceUuid:
     workspace.uuid, roomUuid: roomUuid, startDate: moment(calendarZone.start).
     unix(), endDate: moment(calendarZone.end).unix() })
 else data = await reservationsOfUser( {user:currentUser, startDate: moment(
     calendarZone.start).unix(), endDate: moment(calendarZone.end).unix()})
```

Listing 6.15: getEvents-Funktion

Danach wird eine Schleife der Daten, die vom Backend kommen, durchgeführt und in ein EventInput umgewandelt, damit der Kalender diese anzeigen kann. Die Schleife ist im Codeblock 6.16 zu sehen. Auch hier wird wieder unterschieden, ob die Daten eines Users oder alle eines Arbeitsplatzes angezeigt werden. Die Beschriftung verändert sich dahingehend, dass beim Arbeitsplatz nur die Zeit angege-

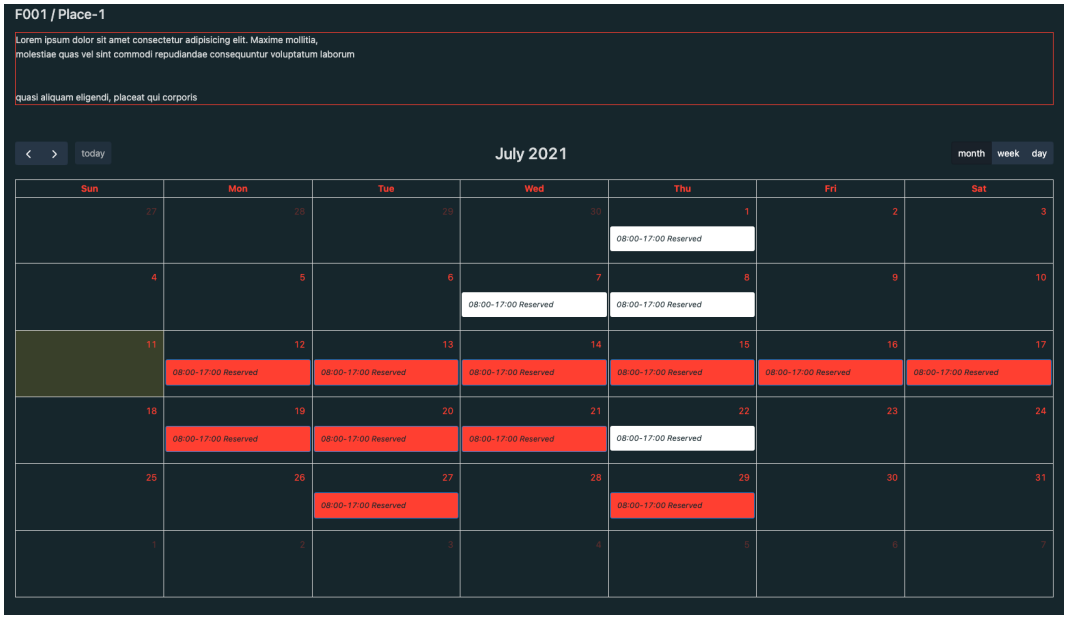

Abbildung 6.1: Kalender

ben wird und dieser dann reserviert ist. Beim Benutzer wird zusätzlich zum Zeitraum der Raum- und der Arbeitsplatzname mit angegeben.

Zusätzlich wird noch geschaut, ob die Reservierung mit dem eingeloggten Benutzer übereinstimmt. Wenn das der Fall ist, ist das Event im Kalender weiß, wenn das nicht der Fall ist, ist es rot, zu sehen in Abbildung 6.1.

```
1  for ( const event of data) {
2    let title: string
3    if (workspace && roomUuid) title = `${moment.unix(event.startDate).format(
         formatHours)}-${moment.unix(event.endDate).format(formatHours)} Reserved`
4    else title = `${moment.unix(event.startDate).format(formatHours)}-${moment.unix(
         event.endDate).format(formatHours)} ${event.roomName}-${event.workspaceName}`
5    currentEvents.push({
6      title,
7      display: `/workspace/${event.roomUuid}/${event.workspaceUuid}`,
8      className: currentUser.uuid === event.userUuid ? 'ownEvent' : 'otherEvents',
9      id: event.uuid,
10     start: moment.unix(event.startDate).toISOString(),
11     end: moment.unix(event.startDate).toISOString()
12   })
13 }
14 setCurrentEvents(currentEvents)
```

Listing 6.16: Umwandlung von Reservierung aus dem Backend in Kalender-Events

Wenn man auf einen Tag klickt, erscheint ein Modal, in dem man Start- und Endzeit eingeben muss, um eine Reservierung abzuschließen. Standardgemäß ist die

Startzeit bei 8 Uhr und die Endzeit bei 17 Uhr. Dieses Modal ist eine selbst erstellte Komponente, welche auch zur Reservierung bei der Mobilen-Ansicht dient.

6.7.2 Kalender Mobile-Ansicht

Da es unmöglich war, den Kalender von der Fullcalendar-Bibliothek benutzerfreundlich auf kleineren Displays dazustellen, musste eine komplett neue Kalender-Bibliothek konzipiert werden. Die revo-calender-Bibliothek [Mol20] war die beste Alternative, da sie am besten funktioniert hat und auch für die Applikation passend ist.

Da der Kalender keine passende Funktion zurückgibt, wann er die Reservierung zum Laden braucht, musste ein useEffect-Hook benutzt werden. Dieser registriert die useState-Hooks month und years [6.17-Z.12]. Wenn sich einer von diesen verändert, wird der aktuelle Monat ausgerechnet, der im Kalender angezeigt wird. Er schickt dann mit dem Monat eine Anfrage an die API. Die Daten werden wieder in ein Event umgewandelt und in den Kalender geladen.

```
1  useEffect(() => {
2    if (month === 0 || year === 0) return
3    (async function getReservationData () {
4      const startDate = moment(new Date(`${year}/${month}/01`)).unix()
5      const newMonth = month + 1
6      const endDate = moment(new Date(`${year}/${newMonth}/01`)).unix()
7      const data: Reservation[] = await mutateAsync({ roomUuid, workspaceUuid:
           workspace.uuid, startDate, endDate })
8      const array: Event[] = []
9      data?.forEach((r:Reservation) => {...})
10     setEventList(array)
11   })()
12 }, [month, year])
```

Listing 6.17: Mobiler Kalender: Daten holen

Die States verändern sich am Anfang, wenn der Kalender initialisiert wird oder wenn der Benutzer den Monat oder das Jahr wechselt.

Die Tage mit Reservierungen sind mit einem Strich markiert. Sobald man auf einen Tag klickt, öffnet sich auf der rechten Seite eine Übersicht mit allen Reservierungen an dem Tag und einer Möglichkeit, eine Reservierung abzuschließen, was auch auf der Abbildung 6.2 zu erkennen ist. Wenn man auf eine Reservierung klickt, taucht ein Löschen-Button auf. Man wird noch mal von einem Modal gefragt, ob man die Reservierung wirklich löschen möchte.

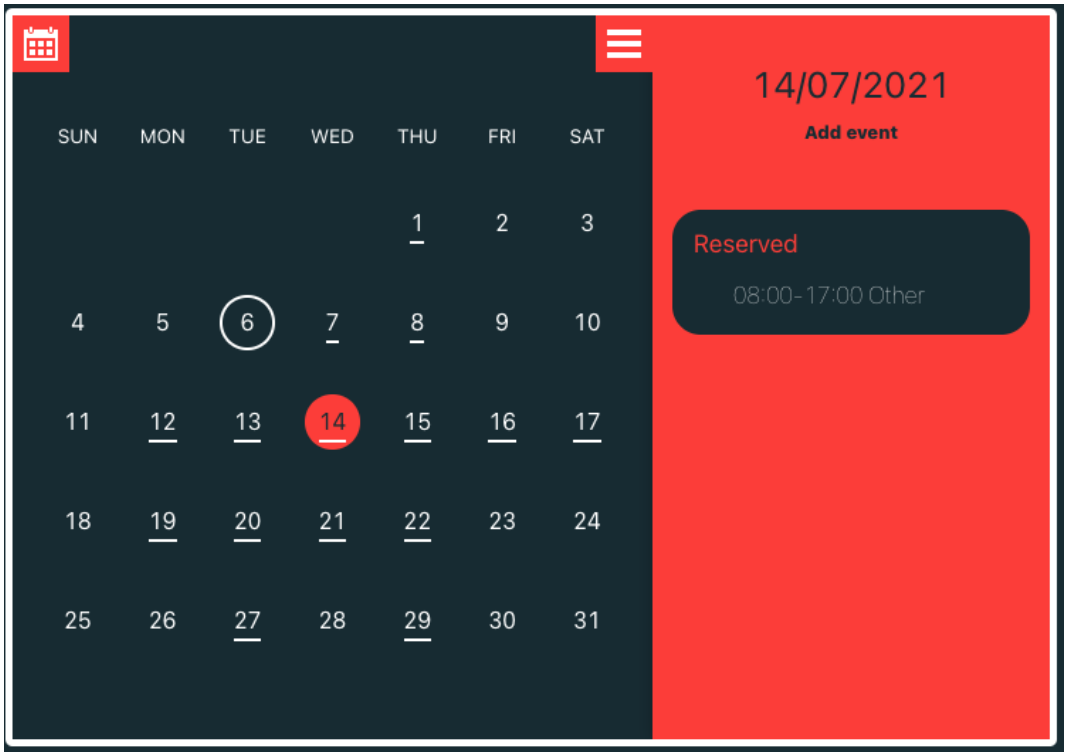

Abbildung 6.2: Mobiler Kalender

6.8 Account-Seiten

Um alle Benutzer anzeigen zu lassen, werden zwei useQuery verwendet. Einmal wird der Endpunkt `/users` angefragt mit den URL-Parametern `page`, `size` und `active` als `false`, um alle nicht aktivierten Benutzer zu bekommen. Bei der zweiten Query werden alle aktivierten Benutzer geholt. Es wird eine Seiten-Variable und eine Größe von 8 Usern übergeben. Sobald die Seiten-Variable verändert wird, wird der useQuery-Hook neu getriggert und die passenden Daten werden zurückgegeben.

Die nicht aktivierten Benutzer werden auf der Account-Listen-Seite oben rechts als Icon und einer roten Zahl angezeigt, wie viel Accounts noch nicht aktiviert sind: zu sehen in Abbildung 6.3.

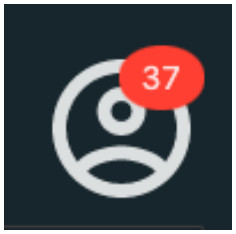

Abbildung 6.3: Nicht aktivierte Accounts

Wenn man auf das Icon klickt, erscheint ein Popup mit allen Usern, welche mit einer Checkbox aktiviert werden können. Auch kann man auf einzelne Benutzer klicken, wobei man dann auf eine Detailansicht weitergleitet wird, auf die gleich noch eingegangen wird.

Die aktivierten Accounts werden wie im Mockup Abbildung 4.2. dargestellt. Die Seitennummerierung ist mit Antd erstellt worden. Sobald sich die Seite verändert, wird mit der pageChange Funktion die Page-Variable verändert, was einen neuen API-Aufruf mit der neuen Seite verursacht. Wenn die Seite schon einmal aufgerufen wurde, wird der Cache benutzt. Die Daten werden nur im Hintergrund neu angefragt, der Admin sieht diesen Vorgang nicht.

```
1 return <Pagination defaultCurrent={currentPage} total={total} onChange={pageChange}
      />
```

Listing 6.18: Seitennummerierung

Wenn man einen bestimmten Benutzer sucht, gibt es die Möglichkeit, nach der E-Mail oder dem Namen zu filtern. Dafür ist das Input-Feld generiert. Sobald in dem Feld etwas steht, wird eine Anfrage mit dem Suchbegriff an die API geschickt. Alle aktivierten Benutzer werden ausgeblendet und das Ergebnis der Suche angezeigt.

6.9 User Detailansicht

Man hat die Möglichkeit, auf jeden Benutzer zu klicken, um seine Detailansicht zu sehen. Die URL der Ansicht ist immer `account/:uuid`. React kann sich die UUID aus der URL ziehen und kann stattdessen direkt vom Backend den Benutzer holen. Bisher ist die Detailansicht nur dafür da, den Benutzer zu verändern. Hierfür muss man auf den Bearbeitungs-Knopf klicken, so hat man die Möglichkeit, Name, E-Mail, Rolle und Passwort zu ändern. Diese wurde mit dem Formular-Element von Antd umgesetzt.

Mit Antd lässt sich einfach ein Formular erstellen, welches nach dem Absenden eine bestimmte Funktion ausführt und alle Werte mit übergibt.

```
1 <Form
2   {... layout}
3     name="change-user"
4     onFinish={save}
5     scrollToFirstError
6   >
7       <Form.Item
```

```
8             initialValue={data.email}
9             name="email"
10            label="E-mail"
11            rules={[
12              {
13                type: 'email',
14                message: 'The input is not valid E-mail!',
15              },
16              {
17                required: true,
18                message: 'Please input your E-mail!',
19              },
20            ]}
21        >
22            <Input />
23        </Form.Item>
24
25  <Form />
```

Listing 6.19: Formular Benutzer-Update

Der Vorteil liegt darin, dass man Regeln für die Felder direkt festlegen und gleichzeitig eine Fehlermeldung mitgeben kann, wenn die Regel gebrochen ist. Die Fehlermeldung wird dann passend unter dem Input-Feld eingeblendet. Zu sehen im Codeblock 6.19 ist das E-Mail-Feld. Es wird festgelegt, dass das Feld ein Pflichtfeld und eine E-Mail sein muss. Hierfür wird in den Regeln bestimmt, dass es vom Typ E-Mail ist, und Antd überprüft diese mit einem passenden Regex automatisch [6.19-Z.13]. Auch kann man eigene Regeln erstellen, wie zum Beispiel beim Passwort, wo zwei Felder gleich sein müssen [6.20-Z.6-14].

```
1   <Form.Item
2       name="password"
3       label="Password"
4       hasFeedback
5       rules={[
6         ({ getFieldValue }) => ({
7           validator(_, value) {
8             if (!value || getFieldValue('confirm') === value) {
9               return Promise.resolve();
10            }
11            return Promise.reject(new Error('The two passwords that you entered do not
                  match!'));
12          },
13        }),
14      ]}
15  >
16      <Input.Password />
17  </Form.Item>
```

Listing 6.20: Eigene Regeln im Formular

Beim Abschicken des Formulars wird die save-Funktion ausgeführt, welche die Daten vom Formular an die API übermittelt. Wieder wird hier useMutation verwendet.

6.10 Arbeitsplan

Zum Schluss wird noch die Arbeitsplan-Komponente beschrieben, die die komplexeste von allen ist, da in dieser viel passieren muss. Zum einen muss diese für den Benutzer den Arbeitsplatz ansehnlich darstellen und zum anderen muss der Admin die Möglichkeit haben, die Tische zu verschieben, die Beschreibung zu bearbeiten oder den ganzen Platz zu löschen.

Damit die Arbeitsplätze verschoben und rotiert werden können, wird die react-moveable-Bibliothek [Mov19] verwendet. In diesem Element lassen sich div-Elemente laden, die bewegt werden sollen. Um die vorhandenen Arbeitsplätze und die Spezifikation dazuzustellen, wird der Raum in der Eltern-Komponente mit übergeben. Jeder Arbeitsplatz wird mit Hilfe der `WorkspaceElement` gerendert.

```
1 <Link
2   to={`/workspace/${roomUuid}/${workspace.uuid}`}
3   className="workspace"
4   data-target={`workspace${index}`}
5   style={{ transform: `translateZ(5px) translateX(${workspace.positionX}px)
        translateY(${workspace.positionY}px) rotate(${workspace.rotate}deg) scaleX(1)
         scaleY(1)` }}
6 >
7   {workspace.name.length <= 14 ? workspace.name : `${workspace.name.substr(0, 14)
        }...` }
8 </Link>
```

Listing 6.21: Arbeitsplatz-Element

Im Codeblick 6.21 ist die Version gezeigt, wenn nur Arbeitsplätze gezeigt werden. Das Element verweist auf die Detailansicht des Platzes [6.21-Z.2]. Mit dem Inline-Style wird die Position des Arbeitsplatzes bestimmt. Die Rotation und die Position sind mit in der Datenbank abgespeichert [6.21-Z.5]. Damit der Name nicht aus dem Link ausbricht, wird noch überprüft, ob die Länge des Namens kleiner gleich 14 Zeichen lang ist [6.21-Z.7]. Die Spezifikation-Komponente sieht so ähnlich aus, hat nur ein anderes Styling und keinen Link, sondern ein Div, da diese ausschließlich für den Arbeitsplan zu Orientierung gilt.

Damit lassen sich Arbeitspläne schön darstellen. Damit die Arbeitsplätze wie richtige Arbeitsplätze aussehen, wurde ein Bild in Photoshop erstellt und in den Hinter-

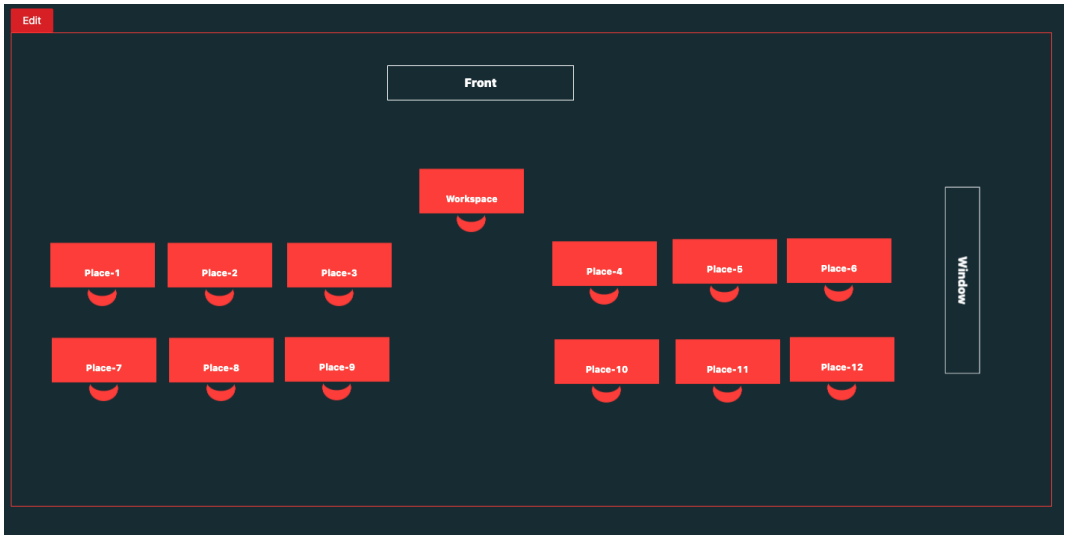

Abbildung 6.4: Raumplan

grund vom Element gelegt, was man in Abbildung 6.4 sehen kann. Auch sieht man den Unterschied zwischen Arbeitsplatz und Spezifikation.

Als Admin hat man die Möglichkeit den Arbeitsplan zu bearbeiten. Dafür dient der Knopf oben links auf Abbildung 6.4. Diesen kann nur der Admin sehen. Wenn man auf diesen klickt, wird die `edit`-Variable im State auf `true` gesetzt und neue Knöpfe erscheinen: Abbrechen, neue Spezifikation hinzufügen, neuen Arbeitsplatz hinzufügen und speichern. Zudem bekommt jedes Element im Arbeitsplan ein Bearbeitungssymbol.

Die Move-Komponente bekommt die `edit`-Variable auch übergeben, damit die Komponente erkennt, wann die Elemente verschoben werden können. Am wichtigsten sind die Parameter `draggable` und `rotatable`, die für die Bewegungen der Elemente verantwortlich sind.

Damit die Komponente erkennt, welches Element bewegt werden soll, muss ein Target übergeben werden. Dies wird mit der OnClick-Funktion geregelt. In dieser werden bei einem Klick auf ein Element die Element-Daten im State der Klasse gespeichert und als Target gesetzt.

```
1 <Move
2     target={target}
3     container={document.body}
4     ref={ref(this, 'move')}
5     keepRatio={edit}
6     origin={edit}
7     draggable={edit}
8     rotatable={edit}
```

```
9     throttleDrag={10}
10    throttleRotate={10}
11    onDragEnd={(e) => this.onEnd(e)}
12    onRotateEnd={(e) => this.onEnd(e)}
13    onRotate={({ target, beforeDelta }) => {
14      // Calc rotate and round it
15      if (!item) return
16      const rotate = Math.round((parseFloat(item.get('rotate')) + beforeDelta) %
            360)
17      target.setAttribute('data-rotate', rotate.toString())
18      item.set('rotate', `${rotate}deg`)
19      target.style.cssText += item.toCSS()
20    }}
21    onDrag={({ target, beforeDelta }) => {
22      if (!item) return
23      let x = parseFloat(item.get('tx')) + beforeDelta[0]
24      let y = parseFloat(item.get('ty')) + beforeDelta[1]
25      if (x <= 0) x = 0
26      if (y <= 0) y = 0
27
28      target.setAttribute('data-x', x.toString())
29      target.setAttribute('data-y', y.toString())
30      item.set('tx', `${x}px`)
31      item.set('ty', `${y}px`)
32      target.style.cssText += item.toCSS()
33    }}
34  />
```

Listing 6.22: Move-Komponente

Sobald man ein Element bewegen möchte, wird die `onDrag`-Funktion ausgeführt. Die Bewegung wird in `beforeDelta` ausgegeben, hiermit lässt sich die neue Position berechnen. Dafür muss der alte x-Wert und y-Wert mit dem neuen Wert aus beforeDelta zusammengerechnet werden [6.22-Z.23-24]. Damit der Arbeitsplan nicht unübersichtlich ist, wird die Bewegung unter 0 nicht mehr verändert [6.22-Z.25-26]. Dies bedeutet, dass sich keine Elemente im Minusbereich ablegen lassen. Zudem wurde `throttleDrag` auf 10 gesetzt und damit erreicht, dass man die Werte nur in 10 Schritten verändert kann [6.22-Z.9]. Das erleichtert es, die Tische ordentlich zu verschieben.

Die Rotation funktioniert ähnlich. Auch hier werden die Rotationen in 10er-Schritten verändert [6.22-Z.10]. Zudem wird der ausgerechnete Wert zusätzlich mit Modulo 360 gerechnet, damit die Rotation nicht ins Unendliche geht und man nur Werte zwischen -360 und 360 bekommen kann [6.22-Z.16].

Sobald man mit dem Rotieren und Bewegen fertig ist, wird automatisch die `onEnd`-Funktion ausgeführt, was in Zeile 6.22-11+12 definiert wird. Diese Funktion upda-

ted die Arrays, mit den neuen Daten. So ist im State immer der aktuelle Stand der Daten vorhanden, die gezeigt werden.

Um einen Arbeitsplatz oder eine Spezifikation zu erstellen, gibt es oberhalb des Plans Knöpfe, die einmal `addRoom` und einmal `addSpec` ausführen. Die Funktionen funktionieren grundsätzlich gleich, aber sie legen in verschiedenen Arrays die Dateien ab. In 6.23 Zeile 2 wird ein neuer Standardarbeitsplatz erstellt, die Positionen auf 0 und der Name auf Workspace gesetzt. Dieses Objekt wird dann in 6.23 Zeile 4 zum Array im State hinzugefügt. Zusätzlich wird ein Modal aktiviert, in dem man den Namen und die Beschreibung bearbeiten kann.

```
1  addRoom = () => {
2    const newWorkspace: Workspace = {uuid: '', name: 'Workspace', positionX: 0,
         positionY: 0, rotate: 0}
3    const workspaceLength = this.state.workspaces.length
4    this.setState({workspaces: [
5        ...this.state.workspaces,
6        newWorkspace
7      ],
8      modalVisible: true,
9      modalDef: 'workspace',
10     modalIndex: workspaceLength
11   })
12 }
```

Listing 6.23: Raum hinzufügen

Im Modal hat man auch die Möglichkeit, das Element zu löschen. Mit Hilfe der `removeElement`-Funktion 6.24 wird das passende Element aus dem Array gelöscht. Hierfür wird im bei einem Klick auf ein Element der Index mitgegeben und im State gespeichert. So kann man leicht mit der JavaScript-Methode splice das Element aus dem Array löschen [6.24-Z.6+10].

```
1  removeElement = () => {
2    let { workspaces, specifications, modalDef, modalIndex } = this.state
3    if (!modalIndex) modalIndex = 0
4    switch (modalDef) {
5      case 'workspace':
6        workspaces.splice(modalIndex, 1)
7        this.setState({ workspaces, target: null, modalVisible: false })
8        break
9      case 'specification':
10       specifications.splice(modalIndex, 1)
11       this.setState({ specifications, target: null, modalVisible: false })
12       break
13   }
14 }
```

Listing 6.24: Element löschen

Wenn man auf einen Speicher-Knopf klickt, wird das komplette Room-Objekt mit den neuen Daten an die API übersendet.

Kapitel 7

Evaluation

Von den funktionalen Anforderungen wurden alle Punkte erfüllt. Hierbei wurden zum Schluss des Projektes im Backend teilweise die Anforderungen mit Tests abgedeckt, die mit dem Paket testing [tes] erstellt wurden. Dieses hilft beim automatischen Testen von Go-Paketen. Mit go test lassen sich alle Funktionen ausführen, die mit Test anfangen, wie zum Beispiel der Test für die Erstellung eines Benutzers.

```
1  func TestCreateUser(t *testing.T) {
2     req := httptest.NewRequest(http.MethodPost, "/api/v1/user", bytes.NewBuffer(
          testUser))
3     w := httptest.NewRecorder()
4     controllers.CreateUser(w, req)
5
6     var m map[string]interface{}
7     json.Unmarshal(w.Body.Bytes(), &m)
8
9     if m["InsertedID"] == nil {
10       t.Errorf("User was not created")
11    }
12 }
```

Listing 7.1: TestCreateUser

In Zeile zwei wird ein neuer Request erstellt, der im Body einen Testbenutzer mitübergibt. Danach wird in Zeile drei die CreateUser-Funktion mit dem Request ausgeführt. Der Test ist erfolgreich, wenn die Funktion ein InsertedID zurückgibt. Wenn das nicht der Fall ist, schlägt der Test fehl.

Die Begründung für die Tatsache, dass nicht alle Anforderungen mit automatischem Test abgedeckt sind liegt darin, dass die Tests erst zum Abschluss implementiert wurden. So lassen sich keine Tests erstellen, in denen es notwendig ist, in der URL UUIDs zu übergeben. Das liegt daran, dass die Endpunkte alle mit mux erstellt wur-

den. Sobald man die URL mit einer UUID in einen neuen Test-Request setzt, kann die getestete Funktion nicht erkennen, ob eine UUID in der URL vorhanden ist. Um dieses Problem zu lösen, müsste man die ganze Endpunkt-Struktur umstellen.

Hätte man bereits zu Beginn die Anforderung gestellt, dass alle Endpunkte mit Tests abgedeckt werden, hätte man von Anfang an die Endpunkte richtig aufgebaut. So hätte man die UUIDs immer aus der URL extrahieren könnten. Noch leichter könnte man es sich mit dem Gin-Gonic-Paket [PB]. Das Gin-Gonic-Paket erleichtert die Tests, gerade bei Autorisierungen und Benutzerverwaltungen, was Kabra [Kab] in seinem Blog veranschaulicht. Mit diesem wäre es ohne Probleme möglich, die Middleware und alle Funktionen zu testen. Aber auch hier müsste man die Endpunkt-Funktion komplett umstrukturieren.

Auch im Frontend lassen sich einzelne Komponenten testen. Dafür wurde das Jest [Fac] Testing-Framework installiert. Mit der React-Testing-Bibliothek [Dod] lassen sich die Komponenten einfach überprüfen. Es wurden drei kleine Tests geschrieben, die veranschaulichen sollen, wie das Testing in React aussehen könnte. Alle Komponenten zu testen, hätte den Rahmen dieser Arbeit überzogen.

```
1  describe('Test Component', () => {
2    it('Render correct', () => {
3      const text = 'Delete Modal'
4      render(<DeleteModal title={text} modalVisible={true} setModalVisible={() => {}}
           deleteFunction={() => {}} />)
5      expect(screen.getByText(text)).toBeTruthy()
6    })
7    it('Cancel Modal', () => {
8      const remove = () => { console.log('Element was deleted correctly') }
9      render(<DeleteModal title="Delete Modal" modalVisible={true} setModalVisible
           ={() => {}} deleteFunction={remove} />)
10
11     const cancelBtn = screen.getByText('Delete')
12     fireEvent.click(cancelBtn)
13   })
14 })
```

Listing 7.2: DeleteModal-Test

Im Codeblock 7.2 wird die DeleteModal-Komponente getestet. In Zeilen drei und neun wird die Komponente zum Testen gerendert. In Zeile vier wird geprüft, ob ein Text gerendert wird. Ist das der Fall, wird die Komponente ausführlich gerendert. Sollte der Text nicht vorhanden sein, würde der Test fehlschlagen und das Modal falsch angezeigt werden. Im zweiten Test wird geschaut, ob die remove-Funktion ausgeführt wird, wenn man auf den Löschen-Knopf drückt. Solange der Test den Knopf drücken kann, schlägt der Test nicht fehl.

Die funktionalen Anforderungen, die nicht mit dem Test abgedeckt wurden, wurden manuell getestet. Alle Endpunkte sind in einer Postman-Datei dokumentiert und können mit Postman getestet werden. Zudem wurde das Frontend sowohl in der Mobilen-Ansicht und als auch in der Desktop-Ansicht in Chrome und Firefox getestet.

Die nichtfunktionalen Anforderungen wurden alle bis auf die Datensicherheit erfüllt [NF105]. Die Sicherheit der Daten ist nicht zu 100 Prozent gewährleistet. Der JWT wird in einem Cookie gespeichert und kann dann bei einer nicht verschlüsselten Anfrage an die API abgefangen werden, wie schon in 5.2.5 Login beschrieben und mit einer Lösung vorgestellt.

Kapitel 8

Schluss

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und die sich daraus ergebenen Erkenntnisse für die weitere Entwicklung des Systems im Ausblick beschrieben.

8.1 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde eine Büroplatz-Buchungsapplikation entwickelt. Die Corona-Pandemie im Jahr 2020/2021 hatte erhebliche Auswirkungen auf die Arbeitsorganisation im Hinblick auf Homeoffice-Tätigkeiten und damit einhergehend die Kapazitäten von Büroarbeitsplätzen in den Unternehmen. Die bereits vorhandenen Möglichkeiten der Buchungsapplikationen können daher noch vielfältig ausgebaut werden.

Die hier entwickelte Buchungsapplikation unterstützt Unternehmen dabei, das hybride Arbeiten umzusetzen. Der Fokus lag darin, eine nutzungsfreundliche Umgebung zu bauen, in der man als Nutzer Reservierungen eines Arbeitsplatzes vornehmen kann und diese dann geordnet angezeigt bekommt. Zudem sollte es eine Rolle geben, die Räume, Arbeitsplätze und Nutzer erstellen oder bearbeiten kann. Zuerst wurden die funktionalen und nichtfunktionalen Anforderungen erstellt. Anhand dieser wurden im Folgenden die Endpunkte und Datenbankmodelle aufgebaut. Für die API wurde die Programmiersprache Go und als nicht relationale Datenbank, MongoDB genutzt. Für das Frontend wurde die Bibliothek React verwendet und Typescript zu Hilfe gezogen. Das Ein- und Ausloggen wird mit einem JSON Web Token geregelt, welcher im Cookie gespeichert wird. In diesem Token werden auch

die wichtigsten Daten wie zum Beispiel die Rolle, gespeichert. Dadurch kann eine Middleware im Backend die Rechte schneller und ohne Datenbankabfrage behandeln.

Ein Admin-Benutzer kann die Seite mit neuen Räumen und Arbeitsplätzen pflegen. Hierfür wurde im Frontend ein Baukasten für einen Platzplaner erstellt. In diesem hat man die Möglichkeit, die Arbeitsplätze wie im Büro darzustellen. Die Position der Arbeitsplätze wird in der Datenbank gespeichert.

Ziel der vorliegenden Bachelorarbeit war es, eine benutzerfreundliche Buchungsplattform auf den modernen Endgeräten zu konzipieren. Das Frontend ist gut gelungen. Gerade das Suchen von Räumen und das Erstellen von übersichtlichen Arbeitsplatzplänen, erleichtern die alltägliche Reservierung für den Nutzer. Insbesondere können die Arbeitsplatzpläne durch Verschiebung und Rotation der Plätze selbsterklärend bearbeitet werden. Aufgrund des unabhängigen Back- und Frontends lässt sich die Applikation für jedes Unternehmen individualisieren und erweitern.

Um einen hohen Qualitätsstandard zu erreichen, wäre die Durchführung von automatischen Tests im Frontend und im Backend nötigt gewesen. Dieser Gesichtspunkt ergabe sich jedoch erst im Laufe der Arbeit, so dass es sinnvoller gewesen wäre, von Anfang mit Tests geplant zu haben. Dann wäre schneller aufgefallen, dass der Quellecode im Backend anders aufgebaut werden muss, um automatische Tests durchzuführen.

Aus den Erfahrungen, die anlässlich der Erstellung der Applikation gesammelt wurden, ergaben sich Erkenntnisse, die der Weiterentwicklung des Buchungssystems dienen können.

8.2 Ausblick

In der weiteren Entwicklung der Applikation wäre es von Vorteil, zwei Event-Kalender gegen einen Kalender auszutauschen. Dies führt dazu, dass ein einheitliches Design und mehr Möglichkeiten entstehen, die Reservierungen darzustellen und zu bearbeiten. Außerdem sollte nur ein API-Call für einen Monat gebraucht werden.

Ausgehend von den hier vorgestellten Ergebnissen könnte man fragen, ob die Arbeitsplatzplan-Komponente in mehrere kleine Komponenten aufgeteilt werden soll-

te, damit im weiteren Programmierverlauf Probleme besser erkennbar sind. Es wäre in diesem Zusammenhang lohnenswert zu untersuchen, ob man das moveable-Paket komplett entfernt und ein eigenes System dafür baut, da das Paket viele Funktionen enthält, die nicht gebraucht werden.

Für größere Unternehmen sollte, für die Benutzer weitere Gruppen mit Regeln anlegen zu können. Auch könnte man eine Verbindung zu einem bestehenden LDAP-Server erstellen. Das würde bei großen Benutzerlisten die Verwaltung von Nutzern leichter machen.

In Zukunft könnten Arbeitsplätze mit Equipment abgespeichert werden, damit der Benutzer in einer Liste sehen kann, wie der Arbeitsplatz ausgestattet ist oder sogar nach bestimmtem Zubehör filtern kann.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

[arb21] arboo. *Arbeitsplatz - Buchungssystem für Office 365*. 2021. URL: https://arboo.io/de/arbeitsplatz-buchungssystem-fuer-office-365/ (besucht am 21. 06. 2021).

[Aut21] Auth0. *JSON Web Tokens*. 2021. URL: https://jwt.io/ (besucht am 10. 07. 2021).

[BB08] Robert Battle und Edward Benson. „Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)“. In: *Journal of Web Semantics* 6.1 (2008). Semantic Web and Web 2.0, S. 61–69. DOI: https://doi.org/10.1016/j.websem.2007.11.002. URL: https://www.sciencedirect.com/science/article/pii/S1570826807000510.

[Daz12] Michael Dazer. „RESTful APIs - Eine Übersicht“. In: (2012). URL: http://snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/restful-apis_dazer.pdf.

[Deu20] EY Deutschland. *90 Prozent der Büroangestellten in Deutschland möchten ihr Homeoffice auch künftig nicht missen*. 2020. URL: https://www.ey.com/de_de/news/2020/12/ey-re-transformation-buerowelten-2020 (besucht am 24. 06. 2021).

[Dod] Kent C. Dodds. *React Testing Library*. URL: https://testing-library.com/docs/react-testing-library/intro/ (besucht am 05. 08. 2021).

[Fac] Facebook. *Jestjs*. URL: https://jestjs.io/ (besucht am 05. 08. 2021).

[fle] flexopus. *Desk Sharing mit dem Flexopus Arbeitsplatz Buchungssystem*. URL: https://flexopus.com/ (besucht am 24. 06. 2021).

[Ful21] FullCalendar. *FullCalendar*. 5. Juni 2021. URL: https://fullcalendar.io/ (besucht am 22. 11. 2009).

[Gac15] Cory Gackenheimer. *Introduction to React*. 1st. USA: Apress, 2015.

[GoL21] GoLang. *Frequently Asked Questions*. 2021. URL: https://golang.org/doc/faq (besucht am 10. 07. 2021).

[Goo09] Google. *Hey! Ho! Let's Go!* 10. Okt. 2009. URL: https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html (besucht am 10. 07. 2021).

[Inc17] MongoDB Inc. *mongodb/mongo-go-driver*. 8. Feb. 2017. URL: https://github.com/mongodb/mongo-go-driver (besucht am 10. 07. 2021).

[Inc21] MongoDB Inc. *MongoDB Clusters*. 2021. URL: https://www.mongodb.com/basics/clusters (besucht am 10. 07. 2021).

[Kab] Kulshekhar Kabra. *Test-driven Development of Go Web Applications with Gin*. URL: https://semaphoreci.com/community/tutorials/test-driven-development-of-go-web-applications-with-gin (besucht am 05. 08. 2021).

[Kam+14] Maulik R. Kamdar u. a. „ReVeaLD: A user-driven domain-specific interactive search platform for biomedical research". In: *Journal of Biomedical Informatics* 47 (2014), S. 112–130. DOI: https://doi.org/10.1016/j.jbi.2013.10.001. URL: https://www.sciencedirect.com/science/article/pii/S1532046413001536.

[Ltd] Condeco Group Ltd. *Arbeitsplatzbuchungssoftware*. URL: https://www.condecosoftware.com/de/produkte/arbeitsumgebungen/ (besucht am 21. 06. 2021).

[Man21] Philipp Mandler. *React. Warum das Webframework für JavaScript so gut ist*. 2021. URL: https://www.micromata.de/blog/react/ (besucht am 24. 06. 2021).

[Mat12] Co. Matt Silverlock. *gorilla/mux*. 2. Okt. 2012. URL: https://github.com/gorilla/mux (besucht am 10. 07. 2021).

[Med20] ParTech Media. *Authentication using JSON Web Tokens in .NET Core*. 8. Apr. 2020. URL: https://www.partech.nl/nl/publicaties/2020/04/authentication-using-json-web-tokens-in-net-core# (besucht am 10. 07. 2021).

[mob] mobiscroll. *Modern React event calendar*. URL: https://demo.mobiscroll.com/react/eventcalendar (besucht am 22. 07. 2021).

[Mol20] Gabriel Molter. *RevoCalendar*. 13. Juli 2020. URL: https://gjmolter.github.io/revo-calendar/ (besucht am 10. 07. 2021).

[Mov19] Moveable. *moveable*. 5. Juli 2019. URL: https://github.com/daybrush/moveable/tree/master/packages/react-moveable (besucht am 10. 07. 2021).

[MS14] Katja Malvoni und Josip Knezovic Solar Designer. „Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware“. In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/malvani.

[MTI21] Stefan Müller, Alexander Klier Tanja Carstensen und Sandra Mierich. Ingo Matuschek. *HOMEOFFICE IN ZEITEN DER PANDEMIE*. 2021. URL: https://www.mitbestimmung.de/html/homeoffice-in-zeiten-der-pandemie-16618.html (besucht am 15. 06. 2021).

[Nel20] Saskia Grote Nele Schwarz. *Mobile First Design – Es ist Zeit umzudenken*. 2020. URL: https://www.meltwater.com/de/blog/mobile-first (besucht am 24. 06. 2021).

[Pat17] Sanjay Patni. *Pro RESTful APIs*. Jan. 2017. DOI: 10.1007/978-1-4842-2665-0.

[PB] Javier Provecho und Bo-Yi Wu. *Gin Web Framework*. URL: https://pkg.go.dev/github.com/gin-gonic/gin#section-readme (besucht am 05. 08. 2021).

[Plo+18] Aaron Ploetz u. a. *Seven NoSQL Databases in a Week: Get up and Running with the Fundamentals and Functionalities of Seven of the Most Popular NoSQL Databases*. Packt Publishing, 2018.

[Rea21] React. *Thinking in React*. 2021. URL: https://reactjs.org/docs/thinking-in-react.html (besucht am 24. 06. 2021).

[tes] testing. *testing*. URL: https://pkg.go.dev/testing (besucht am 05. 08. 2021).

[Yel17] Naren Yellavula. *Building RESTful Web Services with Go: Learn How to Build Powerful RESTful APIs with Golang That Scale Gracefully*. Packt Publishing, 2017.