Studienarbeit

im Fach Praktikum Softwareentwicklung

Task Management-System –

mit Raumbuchungssystem

Hochschule
Hof
University of
Applied Sciences

Verfasser: ***Sebastian Bär***

00279720

Gruppe: “Bottlenecks”

Sommersemester 2022

Hochschule Hof

Dozent: *Stefan Müller*

Abgabetermin: *08.07.2022*

1. Vorwort

Für das Modul “Software-Praktikum” soll ein Softwareprojekt als Studienarbeit umgesetzt werden.
Ziel des Moduls ist es eine Software systematisch und im Team zu erstellen. Dabei sollen agile Software Methoden für die Zusammenarbeit im Team verwendet werden.
Außerdem soll ein Tool zur Versionsverwaltung verwendet werden, in diesem Fall wurde als Vorgabe für das Projekt “Github” gewählt.

2. Planung

2.1 Gruppenbildung

Als Gruppe wurde anfangs eine Gruppe bestehend aus David Weiß, Dejan Fraas und Sebastian Bär mit dem Gruppennamen “Bottlejobs” geformt. Leider musste diese Gruppe aufgrund der Krankheit eines Mitglieds aufgelöst werden und der Rest der Gruppe hat sich einer anderen Gruppe angeschlossen. Die neue Gruppe “Bottlenecks” bestand dann aus den Teammitgliedern Dejan Fraas, Johannes Matus, Daniel Vogel, Eugen Kudraschow und Sebastian Bär.

2.2 Themenauswahl

Innerhalb der Gruppe wurde über ein Thema abgestimmt, da alle Gruppenmitglieder schon Probleme bei der Suche nach freien Räumen in der Hochschule Hof hatten, wurde als Thema ein Raumplaner für die Hochschule Hof gewählt.
Nach der Zusammenführung der Gruppen musste das Thema jedoch abgeändert werden, deshalb wurde in das Task Management-System der anderen Gruppe ein Raumplaner integriert, deshalb die Wahl des Themas “Task Management-System –mit Raumbuchungssystem”.

2.3 Auswahl der Technologien und Frameworks

Als Framework für das Backend wurde Laravel gewählt. Laravel bietet verschiedene Vorteile, der wichtigste Aspekt ist die gute Dokumentation, die gerade für Einsteiger in der Webentwicklung den Umgang mit dem Framework erleichtert. Laravel bietet außerdem einfache Möglichkeiten zur Autorisierung und lässt sich mit der CLI artisan unkompliziert bedienen.

Für das Frontend wurde sich für React als Framework entschieden. React ist sehr intuitiv und flexibel und deshalb ideal für das Projekt und die Entwickler geeignet. React ist zudem ein sehr weit verbreitetes Framework und bietet deshalb viele Quellen und Dokumentationen.

Beide Frameworks können über eine REST-API als Schnittstelle miteinander interagieren, dadurch ist es möglich Front und Backend getrennt zu betrachten und die Entwicklung von beidem parallel zu betreiben.

2.4 Gruppenzusammenführung

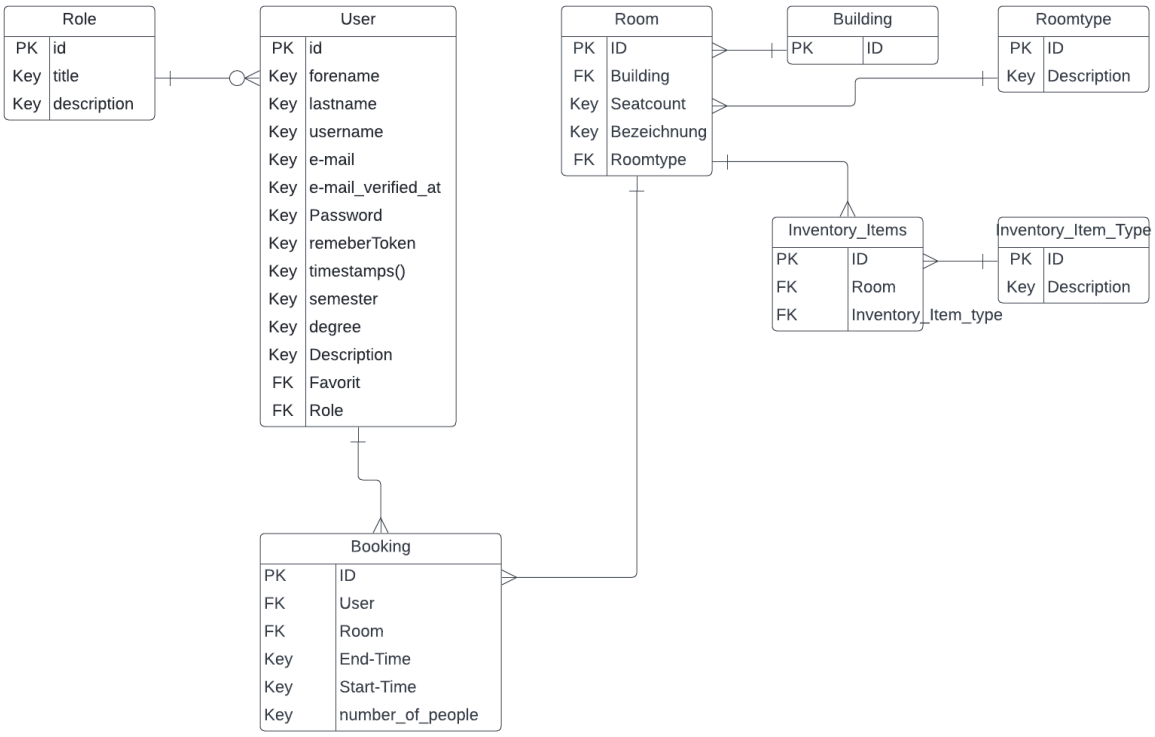
Wie schon im Kapitel 2.1 erwähnt mussten sich zwei Mitglieder der Gruppe “Bottlejobs” der Gruppe “Bottlenecks” aufgrund eines krankheitsbedingten Ausfalls anschließen.
Diese Situation hatte einige Änderungen für das Projekt zur Folge. Beide Gruppen hatten die gleichen Frameworks gewählt, somit war die Umstellung für alle Mitglieder relativ unproblematisch. Kenntnisse bezüglich der beiden Frameworks die bis zu diesem Zeitpunkt erworben wurden, waren deshalb auch in der neuen Gruppenkonstellation von Nutzen.
Beide Gruppen hatten jedoch zum Zeitpunkt der Zusammenführung ihre Design und Entwurfsphase abgeschlossen, dies hatte zur Folge das ein überarbeitetes Design (weiteres dazu im Kapitel 3.2 Design) und Datenmodell (Kapitel 3.1) erstellt werden musste. Auch der bis zu diesem Zeitpunkt erstellte Code musste auf das neue Projekt angepasst werden.

Die neue Gruppenzusammenstellung hatte folgende Aufgabenverteilung zur Folge:
Johannes Matus (Backend), Sebastian Bär (Backend), Dejan Fraas (Frontend), Eugen Kudraschow (Frontend) und Daniel Vogel (Scrum Master/ Frontend)

3. Design und Entwurf

3.1 Datenmodell

Um die Daten des Projektes einfach und sinnvoll zu verwalten wurde ein ER-Modell entworfen. Wie schon in 2.4 erwähnt musste das Datenmodell später aufgrund der Gruppenzusammenführung angepasst werden. Da die meisten Elemente für das zusammengeführte Projekt von der Gruppe “Bottlenecks” erstellt wurde, musste das andere ER-Modell gekürzt wurden. Die wichtigsten Elemente des alten Diagramms sind jedoch auch im neuen wiederzufinden.



Jeder User hat hier eine Rolle, diese Rolle kann entweder Student, Dozent oder Admin sein. Somit könnte man aufgrund dieser Rolle ein Rechtesystem entwerfen, das zum Beispiel Dozenten bei der Raumbuchung vor Studenten bevorzugen würde.

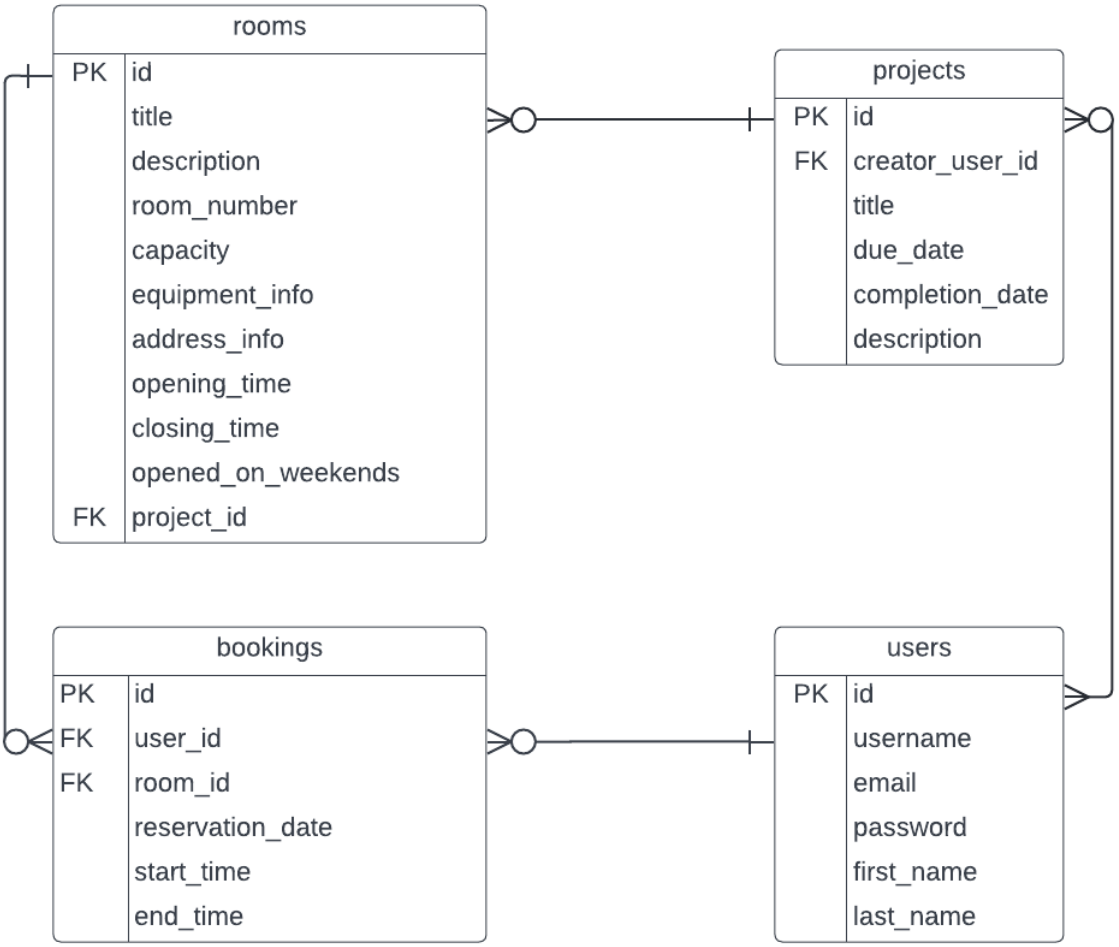
Der User besitzt verschieden Attribute, zusätzlich zu den normalen Personendaten die eine Person besitzt hat diese Tabelle einige Laravel spezifische Attribute. Dies ist unter anderem ein Feld zur Überprüfung der E-Mail Verifikation, das einen timestamp beinhaltet der Auskunft darüber geben kann ob ein User verifiziert ist. Zusätzlich kann der User noch einen Raum favorisieren um auf der Startseite diesen Raum direkt zu sehen.

Der Raum an sich befindet sich einem einem Gebäude (A,B,...) und hat einen bestimmten Typ, zum Beispiel Computerraum.

Zusätzlich gibt es noch Items, diese können einem Raum beinhalten und sind von einem speziellen Typ. So kann man zum Beispiel Räume suchen die ein Whiteboard besitzen.

Das wichtigste ist die Buchung an sich, wichtig dabei ist der Zeitraum der Buchung und die Anzahl von Personen, dies hilft es Überbuchungen zu vermeiden. Dadurch ist es zudem Möglich in der Übersicht die Auslastung der Räume anzuzeigen.

Viele Elemente musste bei der Gruppenzusammenführung (Kapitel 2.4) gestrichen werden, da der Raumplaner nur noch ein Teil des Task Managements wurde und auf die Besonderheiten die ein Raumbuchungssystem für die Hochschule beinhaltet verzichtet wurde.



Nach der Zusammenführung der Gruppen wurden das ursprüngliche Diagramm des Task Managements um die Buchungen und Räume erweitert.

Dabei hat ein Raum zusätzlich zu den ursprünglichen Attributen auch eine Öffnungs- und Schließungszeit. Außerdem hat der Raum die Items direkt als String, somit ist es zwar nicht mehr möglich nach diesen zu sortieren, es ist aber dafür einfacher zu verwalten. Ein Raum wird auch immer einem Projekt zugeordnet, somit können diesen alle Projektmitglieder auch buchen.

Die Buchung an sich ist gleich geblieben, die Anzahl der Personen wurde dafür rausgenommen, da es immer nur eine Buchung für einen Zeitraum geben kann und deshalb es nicht mehr nötig ist die Auslastung zu kontrollieren.

3.2 Design

Im Zuge der Designphase des Projektes wurden Wireframes erstellt.
Diese Wireframes waren ein zentraler Teil des Projektes deshalb wurde diese in Zusammenarbeit aller Teammitglieder erstellt. Die in der Gruppe “Bottlejobs” erstellten Wireframes mussten bei der Gruppenzusammenführung eingebunden und angepasst werden, dennoch musste durch das unterschiedliche Design der beiden Applikationen das

Design komplett an das neue Projekt angepasst werden, dieser Prozess verzögerte den Zeitraum der Designphase.

4. Backend-Entwicklung

4.1 Seeder und Factories

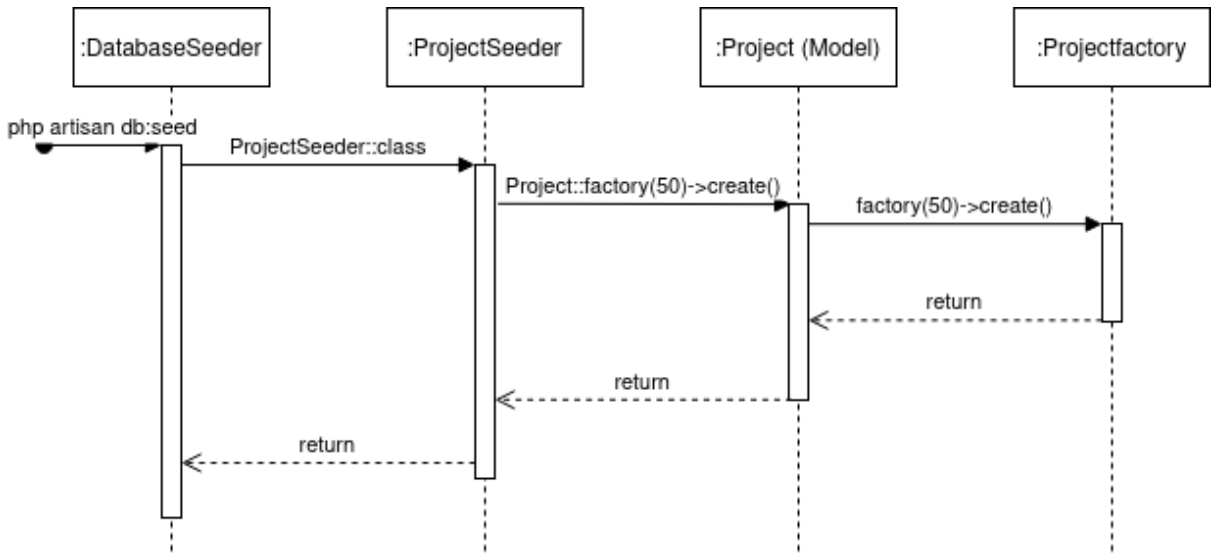
4.1.1 Seeder

Seeder sind ein wichtiger Teil des Projektes und ermöglichen es vordefinierte Daten und über Factories erstellte Objekte auf der Datenbank zu speichern. Voraussetzung für die Seeder sind erstellte Migrationen und Modelle, diese wurden von Johannes Matus auf Basis des ER-Diagramms (Kapitel 3.1) erstellt.

Ein Seeder wird über das CLI Artisan mit dem Befehl php artisan db:seed ausgeführt, um den Seeder und die Migrations gleichzeitig auszuführen kann auch das Kommando php artisan migrate:fresh –seed ausführen.
Grundlage des Seeders ist die Klasse *DatabaseSeeder*, dieser Seeder kann dann alle weiteren Seederklassen aufrufen, somit wird gewährleistet dass alle gewünschten Datensätze erstellt werden.

Der Seeder eines Models wird nur einmal ausgeführt, deshalb macht es Sinn dort nur statischen Daten zu verwenden. Wenn Attribute eines Modells dynamisch erstellt werden sollen, macht es mehr Sinn dieses über eine Factory zu erstellen, die Factories müssen dennoch vom Seeder aufgerufen werden.

Sollen jetzt zum Beispiel 50 Projekte erstellt werden macht es am meisten Sinn vom Database Seeder den Projectseeder aufzurufen und von dort dann die Factory. Ein Beispiel für den Ablauf beim Seeden ist im nachfolgenden Sequenzdiagramm veranschaulicht.



4.1.2 Factories

Factories können Objekte eines Modells dynamisch erzeugen. Der Haupt Anwendungszweck von Factories ist es Testdaten zu erzeugen um diese entweder in der Entwicklungsumgebung oder innerhalb von PHPUnit Testfällen zu verwenden.

Eine Factory kann mit Artisan erzeugt werden, dabei wird sowohl eine Klasse, die von der Klasse Factory erbt, erzeugt, als auch im Modell markiert, dass dieses eine Factory besitzt. Somit ist es danach einfach möglich über einen Aufrufs des Model Namens und die Methode factory() eine beliebige Anzahl von Objekten dynamisch zu erzeugen.

Um die einzelnen Attribute eines Objekts zu erzeugen gibt es mehrere Möglichkeiten. Die einfachste Möglichkeit ist es über den *“fake helper”* und vordefinierte Methoden die Daten zu erzeugen. Beispielsweis ist es möglich einen zufälligen Satz zu erzeugen, dies geht mit dem *faker* über die Methode *sentence* sehr einfach möglich. Der *faker* bietet viele weitere Möglichkeiten unter anderem ist es damit möglich zufällige E-Mail Adressen zu erzeugen. Der *faker* kann zudem als eine Art Zufallsgenerator agieren und zufällige Werten, Datumsangaben und Zeitangaben erzeugen, für diese ist es auch möglich für die Werte bestimmte Grenzwert Angaben zu bestimmen.
Eine weitere Möglichkeit zur Attributs Erzeugung ist es diese aus anderen Modellen zu holen. Beispielsweise kann man sich aus allen Usern die erzeugt wurden einen zufälligen User holen um mit dessen Attributen ein anderes innerhalb des neuen Objektes zu füllen.
Die letzte Möglichkeit ist es statische Daten zu verwenden, dies ist aber nicht zu empfehlen da dies die Möglichkeiten des Tests beschränkt.

4.2 API

4.2.1 Routen

Damit Frontend und Backend miteinander interagieren können, wurden innerhalb des Backends verschiedene Routen angelegt um mithilfe der REST Kommandos Operationen ausführen zu können. Dabei wird zwischen den verschiedenen Kommandos POST, GET, PUT und DELETE unterschieden. Bei einer neu angelegten Route muss deshalb auch das Kommando angegeben werden. Zusätzlich wird bei einer neuen Route auch immer der zu verwendete Controller und Methode angegeben. Routen können auch mit Variablen verwendet werden um leichter zielgerichtete Operationen ausführen zu können.
Ein Beispiel für eine Route wäre zum Beispiel das holen aller Buchungen eines Nutzer, dafür wird in Laravel ind er api.php folgende Zeile hinterlegt:

Route::get('/user/{user_id}/bookings',[\App\Http\Controllers\API\UserBookingController::class ,'show']);

Hervorzuheben sind dort die Verwendung der Operation *get* für die Route aber auch die Benutzung der Variablen *user_id* und der Methode *show* innerhalb des Controllers

Um alle Operationen für einen Controller und Route zu verwenden gibt es zudem die Möglichkeit über die *apiRessource* es festzulegen . Dies wird dadurch gewährleistet, dass beim Erstellen von Controllern 4 Methoden vorgegeben werden.
Die Methode *index* dient dazu eine GET Operationen über die Route auszuführen, dabei werden keine weiteren Parameter übergeben.
Anders gestaltet sich dies bei der *show* Methode, bei dieser kann zusätzlich ein weiterer Parameter übergeben werden, deshalb wird zum Beispiel beim Aufruf der Route *url/api/bookings/1* die Buchung mit der ID 1 ausgegeben, obwohl bei der Routendefinition nur *url/api/bookings* definiert wurde.
Für die DELETE Operation gilt das gleiche Prinzip. Für PUT und POST wird eine Request für die Methoden *store* und *update* verwendet (mehr dazu im Kapitel 4.2.2 Request), die Route bleibt dabei aber gleich.

4.2.2 Requests

Eine Request dient dazu bei der Übermittlung von Daten an die API sicherzustellen, dass diese sich ein einer weiter verarbeitbaren Form befinden. In diesem Projekt war es nötig bei POST und PUT Operationen jeweils eine Request dafür zu definieren. In der Request kann festgelegt werden welchen Namen ein Attribute hat, um eindeutig festzustellen wo es zuzuordnen ist. Außerdem kann festgelegt werden ob der Wert notwendig ist, bedeutet wenn dieser fehlen sollte wird eine Fehlermeldung ausgegeben und die Anfrage wird nicht weiterverarbeitet. Des Weiteren kann der Werttyp und ein Maximum festgelegt werden. Es gibt auch die Möglichkeit die Attribute von anderen Objekten zu verwenden und vorher zu prüfen ob dieses Objekt existiert. Auch das prüfen eines Datumsformat ist möglich.
Ein Beispiel hierfür bietet die *StoreBookingRequest*, bei dieser werden die Daten aus dem Frontend für die Erstellung einer Buchung geprüft:

```
'room_id' => 'required|exists:rooms,id',
'user_id' => 'required|exists:users,id',
'reservation_date' => 'required|date_format:Y-m-d',
'start_time' => 'required|date_format:H:i:s',
'end_time' => 'required|date_format:H:i:s|after:start_time',
```

4.2.3 Autorisierung

Die Autorisierung ist ein wichtiger Aspekt des Projekts. Durch die Autorisierung wird gewährleistet, dass angemeldete Nutzer nur Ressourcen verwenden können, auf die diese auch Zugriff haben sollen.
Um dies zu gewährleisten können in Laravel Policies angelegt werden, diese können auch mit einem Modell verknüpft werden um diese einfacher zu verwenden.
Eine Policy besteht immer aus den Methoden *viewany*, *view, create,* update und forcedelete. Wie schon an der Namensgebung erkenntlich sind diese Methoden dazu geeignet um bei REST-Operationen die Autorisierung zu übernehmen.
Als Beispiel wird für die Buchung die entsprechenden Berechtigungen geprüft:
viewany ist in diesem Beispiel nicht notwendig, da ein Benutzer nie der Lage sein soll alle Buchungen zu sehen. Der Benutzer kann entweder nur seine eigenen Buchungen sehen oder es kann für eine spezifische Buchung in *view* geprüft werden.

Bei *view* ist wichtig dass immer das Booking an sich mit übergeben wird und außerdem der User, dieser muss aber nicht extra angegeben werden. Da sowohl alle Mitglieder als auch der Admin eines Projektes eine Buchung innerhalb des Projektes sehen können, muss innerhalb der *view* dieses Rechtesystem abgebildet werden. Um dies zu gewährleisten wird über die im User Modell integrierten Methoden *isOwnerOfProject* und *isMemberOfProject* dies geprüft.
Um eine Buchung erstellen zu können muss abgefragt werden ob der Benutzer Teil des Projektes zu dem der entsprechende Raum gehört ist, deshalb wird in der *create* der Raum und der User übergeben. Über die Projekt ID des Raums kann dann herausgefunden werden ob der entsprechende User zu dem Projekt gehört, ist dies der Fall ist der User dazu berechtigt eine Buchung für den Raum zu erstellen.
Wenn ein Benutzer versucht eine Buchung über *update* zu aktualisieren muss geprüft werden ob er entweder der Ersteller der Buchung oder der Admin des gesamten Projektes ist. Der Grund dafür ist das wenn zum Beispiel der Benutzer krank sein sollte oder nicht mehr im Projekt tätig ist, muss der Admin trotzdem die Möglichkeit haben die Buchung zu verändern.
Das gleiche Prinzip wie bei *update* gilt auch für *forceDelete*, wieder hat sowohl der Admin als auch der Ersteller der Buchung die Möglichkeit die Buchung zu löschen.

Das Rechtesystem ist meist ähnlich aufgebaut, um dies zu veranschaulichen folgende Tabelle:

Berechtigung / Modell	viewany	view	create	update	forceDelete
Ankündigung	Projekt-mitglieder	Projekt-mitglieder	Projektadmin, User mit ‘can_create_announcements’	Projektadmin Ersteller	Projektadmin Ersteller
Buchung	Projekt-mitglieder	Projekt-mitglieder	Projekt-mitglieder	Projektadmin Ersteller	Projektadmin Ersteller
Projekt	Projekt-mitglieder	Projekt-mitglieder	Benutzer	Projektadmin	Projektadmin
Raum	Projekt-mitglieder	Projekt-mitglieder	Projektadmin	Projektadmin	Projektadmin
Tag	/	/	Projektadmin, User mit ‘can_create_tags’	/	Projektadmin
Task	Projekt-mitglieder	Projekt-mitglieder	Projektadmin, User mit ‘can_create_tasks’	Projektadmin Ersteller (complete) Ersteller, Zugewiesener Benutzer, Projektadmin	Projektadmin Ersteller

4.2.4 Controller

Der wichtigste Bestandteil der API sind die Controller, innerhalb der Controller wird die Programmlogik des Backends abgebildet. Zu den Routen sollte es jeweils einen passenden Controller geben, so gibt es zum Beispiel für die Route *api/bookings* den *BookingController* aber auch für *api/user/user_id/bookings* den UserBookingController.
Wie schon in Kapitel 4.2.1 erwähnt hat ein Controller meist 4 Methoden die vergleichbar mit den REST-Operationen sind.
Die erste dieser Methode ist die *index* Methode, wie schon vorher erwähnt findet hier keine Übergabe von Parametern statt, somit ist diese Methode ideal dafür geeignet um alle Objekte eines Modells zurückzugeben.
Anders gestaltet sich dies bei der *show* Methode, hier wird meist die ID mit übergeben um ein Objekt zurückzuliefern. Bei allen Methoden im Controller die eine Beschränkung beinhalten sollen ist es sinnvoll zu prüfen, ob diese den Vorgaben gemäß unseres Rechtssystems (Kapitel 4.2.3) entspricht. Hierfür wird ein Aufruf der Methode *authorize* durchgeführt, bei diesem werden dann die gewünschte Methode, in diesem Fall wäre das *view*, mit übergeben und eventuelle Übergabeparameter, in diesem Fall die Buchung.
Bei der Methode *store* wird eine Request (Kapitel 4.2.2) übergeben, diese muss dann erst validiert werden um eventuelle Fehlermeldungen aus den Vorgaben der Request zu Erzeugen. Ist die Anfrage validiert, kann mit den Daten der Anfrage weiter gemacht werden. So ist es zum Beispiel in der Programmlogik bei der Buchung notwendig auf den richtigen Zeitraum der Buchung zu achten, da der Raum Öffnungszeiten besitzt und eine Buchung nur innerhalb dieses Zeitraums gemacht werden kann. Außerdem muss geprüft werden ob andere Buchung in diesem Zeitraum schon existieren. In beiden Fällen sollten Fehlermeldungen ausgelöst werden und aus Rückgabe für das Frontend übergeben werden. Desweiteren wird geprüft ob die Buchung erzeugt worden ist und diese dann zurückgeben.
Beim *update* gilt das gleiche Prinzip wie für *store*, es gibt nur Unterschiede bei der Art der Request, Berechtigung, Methodenaufruf und Fehlermeldungen.
Anderes ist es jedoch bei *destroy*, hier wird nur die ID übergeben. Das Objekt muss also wieder erst gesucht werden, was zu Fehlern führen könnte. Außerdem muss der Löschvorgang erfolgreich abgeschlossen werden und die Berechtigung korrekt sein.

4.2.5 Responses

Wie in vorherigen Kapitel erwähnt gibt es bestimmte Fehlermeldungen aus dem Backend die im Frontend abgefangen werden muss, Grundlage dafür sind die unterschiedlichen Response und Statuscodes.
Eine Response ist immer im JSON-Format und gibt immer zuerst an ob es erfolgreich war oder nicht. Zudem wird eine Nachricht, die entweder ein Objekt oder mehrere Objekte ist aber auch eine Fehlermeldung sein kann. Am wichtigsten hierbei ist der Statuscode, so sind beispielsweise die Codes 200,201 Codes für eine erfolgreiche Verarbeitung im Backend. Dagegen sind die Fehlermeldung 404 (nicht gefunden), 500 (unerwarteter Fehler) oder 403 (nicht berechtigt) hinweise an das Frontend das etwas nicht stimmt. Eine genaue Auflistung der verschiedenen Fehlermeldungen finden sich in der API-Dokumentation des Projektes.

Als Beispiel für den Ablauf beim Erstellen einer Buchung ein vereinfachtes Sequenzdiagramm des Vorgangs innerhalb des Backends: