Studienarbeit

im Fach Praktikum Softwareentwicklung

Task Management-System –

mit Raumbuchungssystem

Hochschule
Hof
University of
Applied Sciences

Verfasser: ***Johannes Matus***

00325520

Gruppe: "Bottlenecks"

Sommersemester 2022

Hochschule Hof

Dozent: Dipl.-Inf. *Stefan Müller*

Abgabetermin: *08.07.2022*

Inhalt

1. Vorbemerkungen

Das Projekt wurde im Rahmen des Faches “Praktikum Softwareentwicklung” von der Gruppe “Bottlenecks” angefertigt. Die Applikation trägt den Namen “Stiva”.
Dieses Dokument dient dazu, einen Überblick über das verwendete Design der Anwendung im Backend zu geben. Sie enthält alle wichtigen Informationen zur Struktur, welche ein Programmierer kennen sollte, bevor er sich in die Software einarbeitet, um Änderungen oder Ergänzungen des Quelltexts vorzunehmen.

Beim im Backend verwendeten Framework, handelt es sich um Laravel (Version 9.11.0). Das Framework wurde dazu eingesetzt um die Datenbankschicht (MySQL) zu abstrahieren und über eine REST-API mit dem Frontend zu interagieren. Die Zugangsdaten zur Datenbank werden in der *.env* Datei konfiguriert.
Da Laravel als Fullstack-Framework die MVC Architektur erzwingt, für Stiva das Frontend allerdings als Single Page Application ausgelagert wurde, wird Laravel hier genutzt um eine API für die jeweiligen Operationen zur Verfügung zu stellen.
Alle Methoden im Quellcode sind mit PHPDoc Kommentaren versehen um die Nutzung und das Verständnis der Klassen zu vereinfachen.

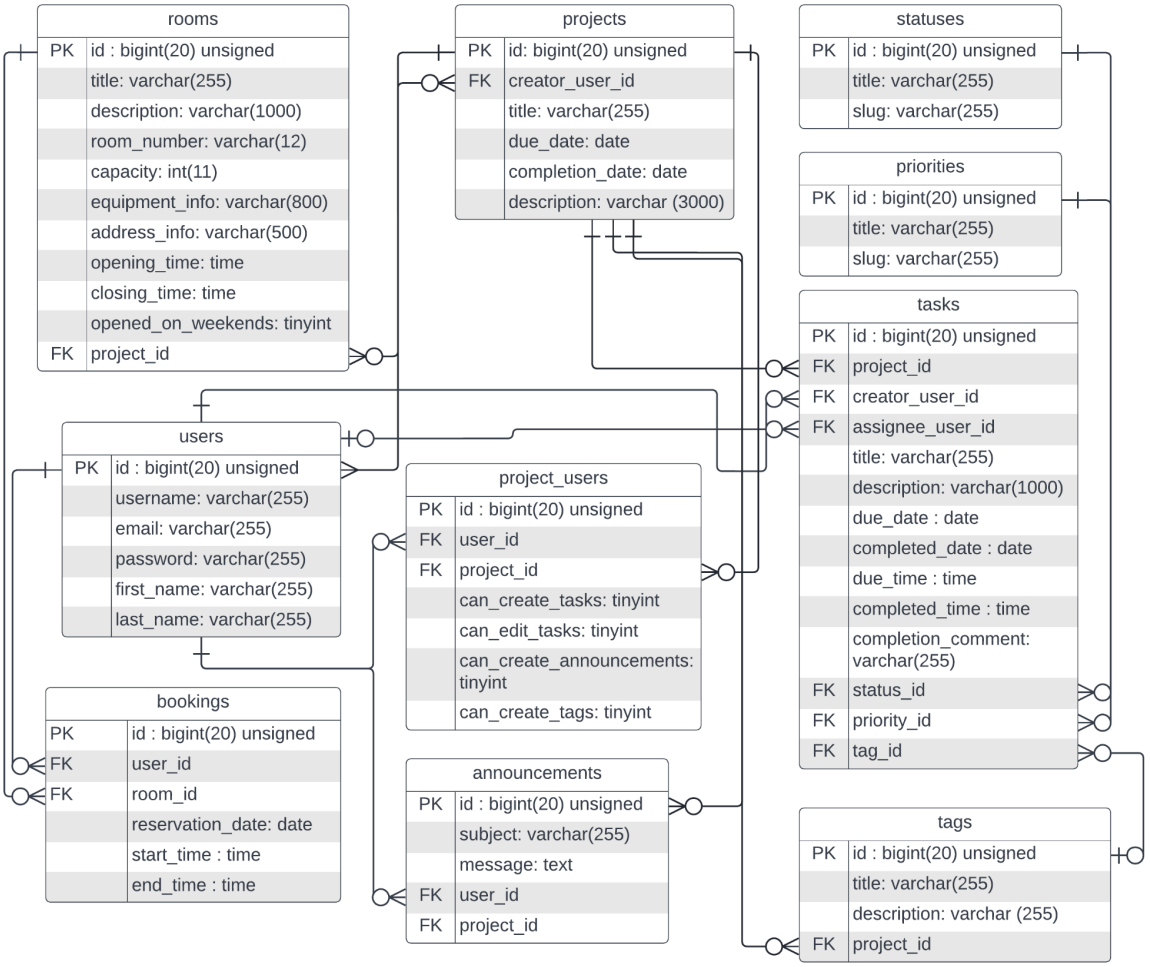
Die Clientseitige Implementierung wird in diesem Dokument nicht näher beschrieben.

Grundlegend stellt das Backend die Möglichkeit zur Verfügung Nutzer zu Registrieren, An- und Abzumelden und dann über die Accounts Projekte zu erstellen. Die Projekte gelten als “Eintrittspunkt” für weitere Funktionen. So kann ein Nutzer innerhalb eines eigenen Projekts weitere Nutzer über den Nutzernamen hinzufügen und deren Rechte verwalten. Innerhalb eines Projekts können Tasks, Tags und Ankündigungen erstellt werden. Veröffentlichte Ankündigungen können von jedem Nutzer innerhalb des Projektes gesehen werden. Erstellte Räume können von den Projektmitgliedern für einen Zeitraum innerhalb eines Tages gebucht werden. Die jeweiligen Aktionen werden durch das Backend ausgeführt, authentifiziert, autorisiert und beantwortet.

2. Datenmodell

2.1. Entity-Relationship Diagramm

Die Applikation baut auf folgendem relationalem Datenmodell auf:



Die ids der jeweiligen Tabellen erhöhen sich alle automatisch bei neuen Einträgen (*auto_increment*).
Zusätzlich zu den aufgeführten Spalten haben alle alle Tabellen noch zusätzlich folgende Spalten (außer: *statuses* und *priorities*):

- created_at
- updated_at

Die Spalten *created_at* und *updated_at* werden von Laravel vorgegeben und werden bei den entsprechenden Operationen zum aktualisieren oder erstellen automatisch

aktualisiert/gesetzt (die Spalten können in Migrationsdateien manuell entfernt werden).
Die Tabelle *ProjectUsers* dient als Zwischentabelle zwischen *User* und *Project*. Hier werden die Projektmitglieder und deren Rechte innerhalb des jeweiligen Projektes gespeichert.
Die *slug* Spalte in *statuses* und *priorities* hat den Zweck, die Einträge auch im Falle einer Änderung des Titels noch für den Menschen lesbar zu identifizieren und zu überprüfen ob der Titel sinnvoll ist.

2.2. Migrationen

In den Migrationsdateien kann das Datenbankschema beschrieben werden, damit Laravel das Erstellen der Datenbanktabellen samt constraints übernehmen kann.
In der up() Methode der Migrationsdateien können die Spalten der Tabelle samt Datentyp und Constraints festgelegt werden (Methode wird über den Befehl *php artisan migrate* ausgeführt).
Die down() Methode soll die up() Methode rückgängig machen (Methode wird über den Befehl *php artisan migrate:rollback* ausgeführt).
Die Migrationsdateien für das Datenbankschema sind im Verzeichnis */database/migrations* gespeichert.

2.3. Models

Die Model-Klassen sind im Unterordner *app/http/Models* zu finden und liegen als Abstraktionsschicht über dem Datenmodell bzw. der Datenbank. Die in den Migrationsdateien festgelegten Spalten müssen in den Model-Klassen nicht explizit als Attribute gesetzt werden, sondern sind über die Datenbankspalten, die sie abbilden, wie “normale” Attribute zugänglich.
Hierbei ist zu beachten, dass die Model Attribute/Spalten über das *fillable* oder *guarded* Array geschützt werden können. *Fillable* beinhaltet Spalten, welche in der Datenbanktabelle eingefügt werden können, *guarded* legt fest, in welchen Spalten nichts eingetragen werden kann. Beim festlegen kann zwischen *fillable* und *guarded* gewählt werden, da sich die beiden gegenseitig bedingen (alle Spalten die nicht in *fillable* beinhaltet sind, sind *implizit* guarded und umgekehrt).

Mit Objekten kann über Laravel Eloquent ORM (object relational mapping) interagiert werden.
Hierzu implementieren die Model-Klassen Methoden, die diese Beziehungen nachbilden. Dazu sind Methoden in der Basisklasse aller Model-Klassen *Illuminate\Database\Eloquent\Model* vorhanden.

Über die Methoden die die Datenbankbeziehungen nachbilden, kann die mit dem Modell verknüpfte Datenbanktabelle abgefragt werden. Dabei lässt sich auf Attribute die über einen Fremdschlüssel referenziert werden, und in einer anderen Tabelle gespeichert sind, zugreifen wodurch keine langen Datenbankabfragen nötig sind.

3. Authentifizierung

3.1. Token basierte Authentifizierung

Die Authentifizierung findet über API-Tokens statt und ist mit Hilfe von *Laravel Sanctum* implementiert, um die Applikation “stateless” zu halten.
Nach erfolgreichem Login, wird dem User ein Bearer Token in der Antwort zurückgeliefert. Dieses Token muss bei jeder Anfrage, für die Authentifizierung notwendig ist, mitgeschickt werden.
Der Nutzer kann so eindeutig identifiziert werden, wodurch die id des Users beim erstellen von Ressourcen automatisch ermittelt und gesetzt wird.
Die Anzahl paralleler Logins ist theoretisch nicht begrenzt.
Beim Logout werden *alle* momentan gespeicherten Token des Benutzers gelöscht.
Die Gültigkeitsdauer der Token beträgt *10 Stunden,* danach muss ein neues Token erstellt werden (erneuter Login erforderlich).
Die Gültigkeitsdauer des Tokens ist in */config/sanctum.php* konfiguriert.
Für die Authentifizierung (Registrierung, Login, Logout.) ist der *AuthController* zuständig.

4. Autorisierung

4.1. Policies

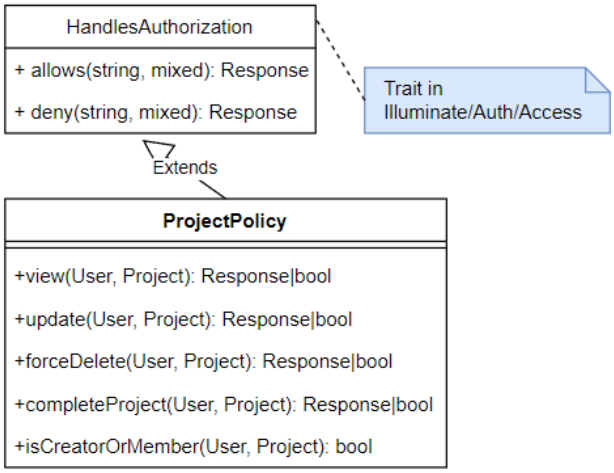
4.1.1. Naming Convention

Policy Klassen sollten mit *<Modelnamen>Policy* benannt werden, damit sie sowohl von Laravel als auch von Entwicklern der passenden Model Klasse zugeordnet werden können.

Sollte die Policy anders benannt sein muss sie in *app/Providers/AuthServiceProvider.php* dem entsprechenden Model zugeordnet werden, damit sie von Laravel erkannt wird.

4.1.2. Funktionsweise

Policies stellen Methoden zur Verfügung, die für das entsprechende Model Berechtigungen des anfragenden Users prüfen. Die *update(...)* Methode prüft so zum Beispiel ob der User ein bestimmtes Model updaten darf. Die erfolgreiche Autorisierung kann durch einen boolean Wert (true → autorisiert, false → nicht autorisiert) bestimmt werden. Sollte eine detaillierte Nachricht mit zurück geliefert werden, kann dies über die *deny(message,code)* Methode der Policy geschehen, die eine *unauthorizedException* wirft und vom in Laravel mitgelieferten Trait *HandlesAuthorization* bereitgestellt wird. Wird die *unauthorizedException* vom Controller nicht behandelt gibt Laravel automatisch eine *403 - Unauthorized* Http-Response zurück.



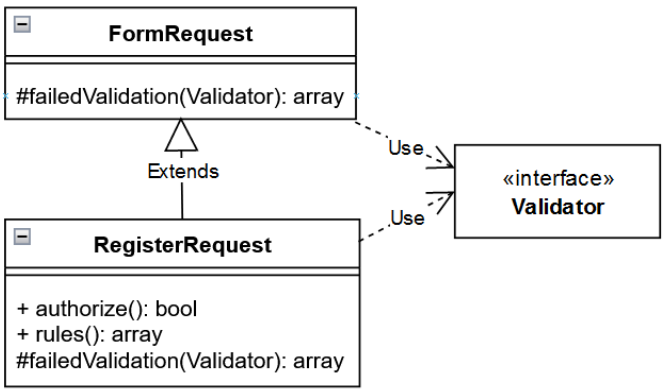
5. API-Endpoints und Controller

5.1. Http Response Codes

200	Erfolgreiche Request
201	Neues Objekt erstellt
400	Bad Request → Liefert JSON Objekt mit Fehlernachricht zurück
401	Der Nutzer konnte nicht authentifiziert werden
403	Der Nutzer ist nicht autorisiert die gewünschte Aktion auszuführen
404	Das Objekt (mit der angegebenen id) existiert nicht
422	Nicht verarbeitete Entität wegen semantischer Fehler (ungültige Daten)
500	Server Error

5.2. Request Klassen

Eigens erstellte Request Klassen sind in */app/Http/Requests* gespeichert und erben standardmäßig von *FormRequest*.



Die Methode *authorize()* muss *true* zurückgeben, damit die Request validiert und an Controller weitergegeben werden kann.

In der Methode *rules()* werden die Regeln zur Verifizierung der gesendeten Daten in einem Array angegeben (Zum Beispiel das Format von E-Mail Adressen, maximale String Längen, notwendige Felder, …).

Da die Methode *failedValidation(Validator $validator)* in *FormRequest* bei einer fehlgeschlagenen Validierung versucht zu einer URL weiterzuleiten, wird die Methode in eigenen Request Klassen überschrieben, um eine *HttpResponseException* zu werfen, die die als fehlerhaft identifizierten Werte der Request als JSON-Antwort zurückliefert.

Die erstellten eigenen Request Klassen können in der Parameterliste der entsprechenden Controller Methode durch type-hinting des Typs ersetzt werden:

*public function store(**StoreModelRequest** $request){...}*
Die Validierung führt Laravel im Anschluss automatisch durch.
Die Namen der Request Klassen sind wie folgt aufgebaut: *<Aktion><Modelname>Request.php*.

5.3. Controller

5.3.1. Allgemeines

Da Laravel das Model-View-Controller Pattern erzwingt, das Frontend aber separat durch eine Single-Page-Application erstellt wurde, sind die Controller dafür verantwortlich eine http-Anfrage anzunehmen, mit dem entsprechenden Model zu interagieren und anschließend eine Antwort zu senden, statt eine neue View.
Der Ablauf für Anfragen ist prinzipiell bei jedem Controller analog.
Standardmäßig stehen 4 Methoden zur Verfügung um die CRUD Operationen für den Endpoint */api/Modelname* auszuführen.
Folgende Methoden sind für die jeweilige http Methode verantwortlich:

- GET - show (einzelne Instanz), index (alle Instanzen der Klasse)
- POST - store
- PUT - update
- DELETE - delete

Weitere Methoden der Controller müssen in *routes/api.php* dedizierte Endpoints erhalten, die nicht mit */api/Modelname* beginnen oder anderweitig von den CRUD-Routen unterscheidbar sind und außerdem die http-Methode, den Controller und die Methode des Controllers spezifizieren, damit Laravel die Anfragen eindeutig zuordnen kann. Die URL des Endpoints wird im Projekt hierbei meistens um ein Schlüsselwort erweitert, um die genauere Aktion zu beschreiben die vom jeweiligen Endpoint ausgeführt wird.

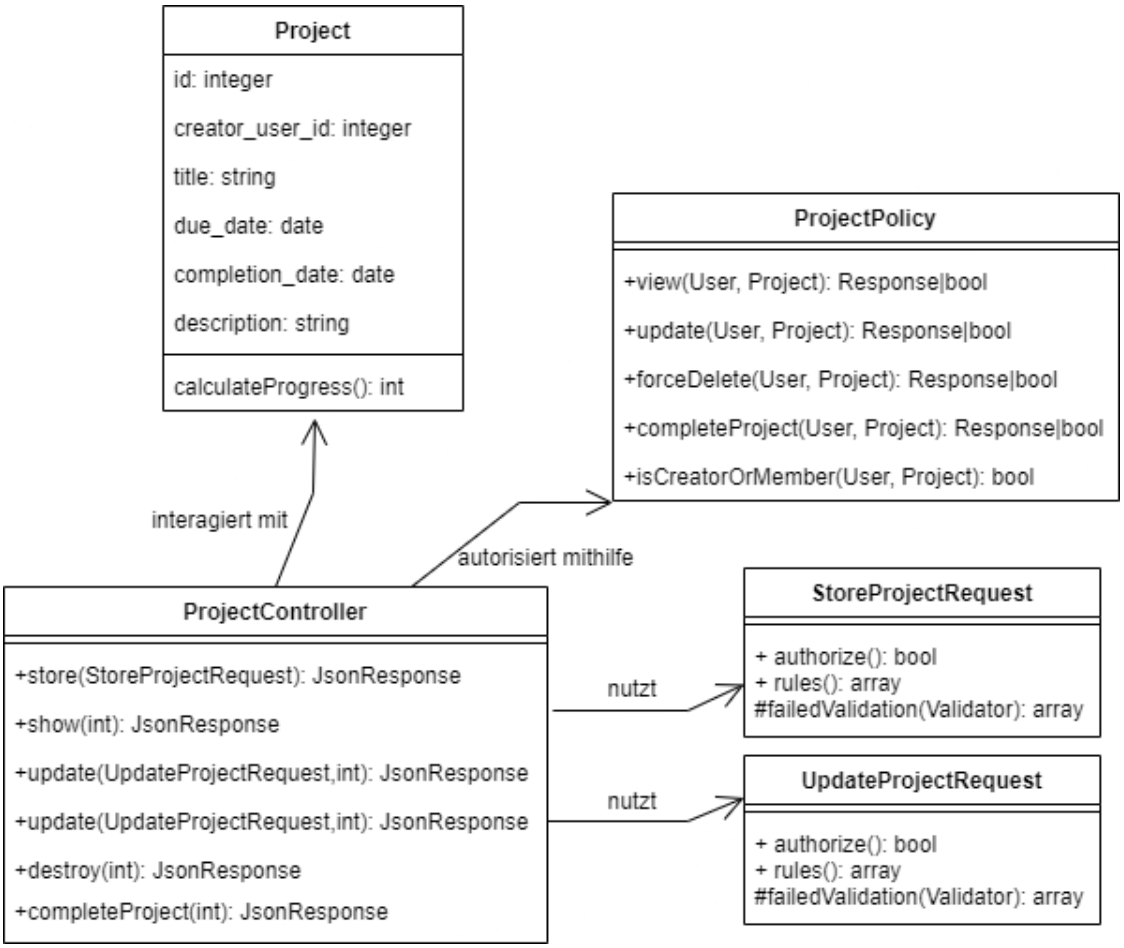
Beispiel:
Methode um ein Projekt abzuschließen: *completeProject(id)* in *ProjectController*

Definition des Endpoints in *api.php*:
*Route::put('**/project/{id}/complete**', [ProjectController::**class**, 'completeProject']);*

Die restlichen CRUD Operationen sind nur auf /api/project/{id} deklariert.

5.3.2. Klassendiagramm und Naming Convention

Die nachfolgenden Beispiele werden anhand der Klasse *ProjectController* den Aufbau der Applikation darlegen. Der *ProjectController* ist dafür zuständig mit Models des Typs Project zu interagieren und stellt dafür verschiedene Methoden zur Verfügung (siehe Abbildung).



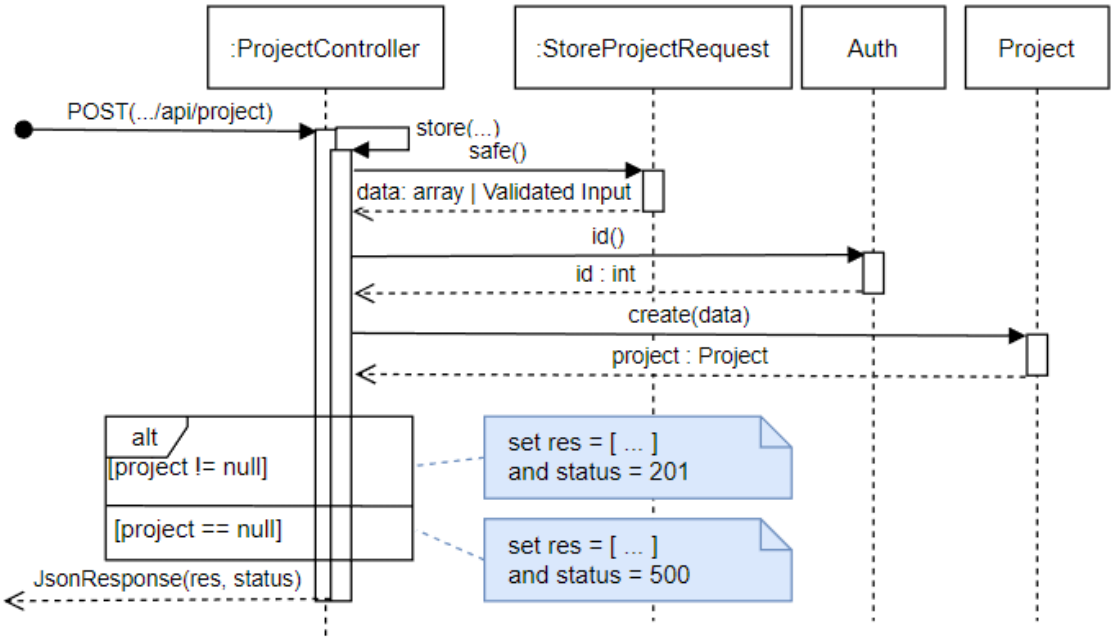
Controller Klassen werden prinzipiell mit *<Modelname>Controller* benannt. Sollten Controller allerdings Aktionen für weitere Modelklassen ausführen oder besondere Aufgaben übernehmen, kann diese Funktionalität auch in eigene Controller Klassen ausgelagert werden um die ursprüngliche Controller Klasse nicht zu überladen.
Neben dem *ProjectController* existieren so zum Beispiel noch folgende Controller für Projekte:

- ProjectOverviewController - Projektstatistik
- ProjectRoomController - Operationen für alle Räume innerhalb eines Projektes
- ProjectTagController - Operationen für alle Tags eines Projektes
- ProjectTaskController - Operationen für alle Tasks innerhalb eines Projektes

Für andere Controller wird dies analog gehandhabt.

5.3.3. Ablauf einer Request zur Projekterstellung

Der nachfolgende Ablauf soll beispielhaft zeigen, wie eine Anfrage verarbeitet wird und findet bei anderen Controller(methode)n bzw. Endpoints **analog** statt.

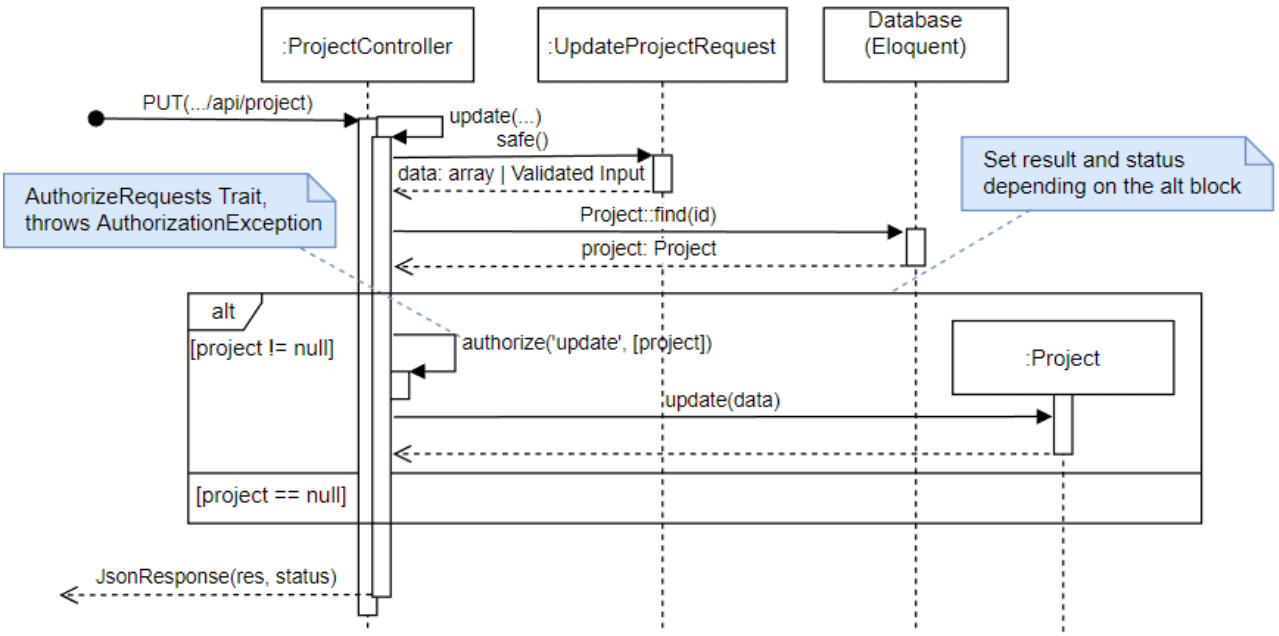


Die Request auf den entsprechenden Endpoint wird automatisch an den Controller weitergeleitet, nachdem die Laravel Sanctum Middleware den User über das Bearer Token authentifiziert hat und die Anfrage durch *StoreProjectRequest* validiert wurde. Im Anschluss daran kann der Controller auf die validierten Daten der Request zugreifen.
Die id des Users, der die Request gestellt hat, wird über die von Laravel bereitgestellte *Auth Facade* ermittelt und im data-Array festgelegt (Voraussetzung hierfür ist die Authentifizierung durch die Sanctum Middleware).

Im Anschluss wird das Projekt mit den validierten Daten und der momentanen user_id erstellt. Nach erfolgreicher Erstellung wird das Projekt samt Http Status Code als Json Response ans Frontend zurückgeschickt.

5.3.4. Ablauf einer Request zur Projektaktualisierung

Bei einigen Operationen ist eine Autorisierung der Anfrage notwendig um sicher zu stellen, dass keine unbefugten Nutzer Daten manipulieren.
Dazu wird vom Controller die Methode *authorize()* aufgerufen. Als Argumente werden die korrespondierende Methode der Policy des Models, für das eine Aktion ausgeführt werden soll, und das konkrete Objekt des Models übergeben.
Die Methode *authorize(...)* wird vom *AuthorizeRequestsTrait* implementiert und von der Basis-Controller-Klasse genutzt.
Die Regeln für die Autorisierung sind in der entsprechenden Policy (in diesem Fall: *ProjectPolicy*) festgelegt.
Die *AuthorizationException* im Falle einer ungültigen Request wandelt Laravel automatisch in eine Antwort um (Http-Statuscode 403).
Ablauf im Sequenzdiagramm:



5.4. API-Dokumentation

5.4.1. Response Aufbau

Neben einem Http-Response Code werden in den Antworten der API immer noch ein *success* Feld und das erstellte/bearbeitete Objekt zurückgegeben. Im Falle eines unbekannten Fehlers oder einer Aktion für die kein zurückliefern des Objektes notwendig ist wird in einem *message* Feld ein Hinweis über die ausgeführte Aktion angegeben.

Der Aufbau der Antworten kann in der API-Dokumentation anhand der Beispielantworten genauer nachvollzogen werden.

Die API - Dokumentation ist ein eigenständiges Dokument das alle verfügbaren Endpoints beschreibt.