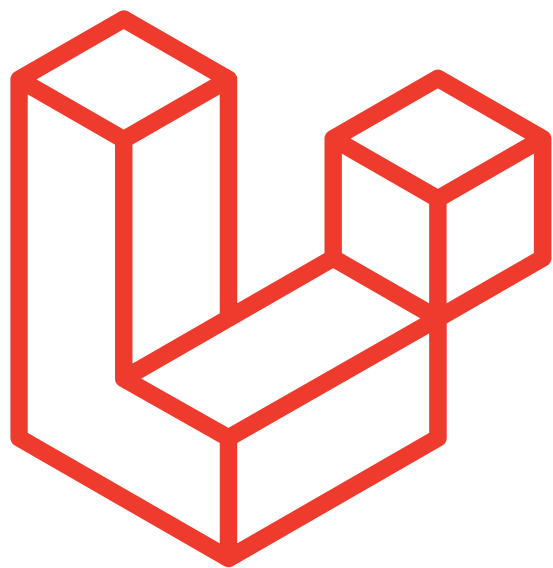


# PROGRAMMIEREN EINER QUIZ WEBSEITE

Mit dem PHP-Framework **Laravel**

***php***



Adrian Schubek

Besondere Lernleistung | Informatik Q2 DA | 2020

# Inhaltsverzeichnis

<b>1   EINLEITUNG .....</b>	<b>3</b>
Vorüberlegungen & Zielsetzung .....	3
<b>2   GRUNDLEGENDES .....</b>	<b>4</b>
2.1   Client-Server Modell .....	4
2.2   Design Patterns.....	5
2.2.1   Model-View-Controller .....	5
2.2.2   Dependency Injection.....	6
2.2.3   Active Record .....	7
2.3   Technologie Stack.....	8
<b>3   LARAVEL .....</b>	<b>10</b>
3.1   Ablauf / Request Lifecycle .....	10
3.2   Service Container .....	11
3.3   Eloquent ORM .....	12
3.3.1   Beziehungen .....	13
3.3.2   Soft Deletes .....	14
3.3.3   Pagination.....	15
3.4   Blade.....	15
3.5   Queue .....	16
<b>4   QUIZ .....</b>	<b>18</b>
4.1   Vorüberlegungen.....	18
4.2   Datenbank .....	19
4.3   Router .....	21
4.4   Middleware .....	22
4.5   Controllers.....	23
4.6   Models.....	25
4.7   Sicherheit .....	26
4.7.1   Validierung.....	26
4.7.2   Authentifizierung .....	27
4.7.3   Autorisierung.....	28
4.7.4   Password Hashing .....	30
4.7.5   Cross-Site-Request-Forgery (CSRF).....	30
4.8   Notifications.....	31
4.9   Front End.....	32

4.9.1   Views.....	32
4.9.2   Livewire.....	34
5   FAZIT .....	35
6   LITERATURVERZEICHNIS .....	35

# 1 | Einleitung

In dieser Besonderen Lernleistung werde ich die Programmierung einer Quiz Webseite ausführlich dokumentieren mit passenden Illustrationen.

## Vorüberlegungen & Zielsetzung

Angemeldete Benutzer der Webseite sollen Quizze spielen und selber erstellen können. Ein Quiz besteht dabei aus mehreren Fragen, welche jeweils bis zu vier Antwortmöglichkeiten enthält wovon genau eine richtig ist. Ebenso können Kommentare für jedes Quiz geschrieben werden.

Sollte dem Benutzer ein Quiz oder Kommentar gefallen hat dieser die Möglichkeit das Quiz/den Kommentar positiv zu bewerten („ liken“). Zusätzlich wird dann der Ersteller des Quiz/Kommentars darüber benachrichtigt.

Unangemeldete Benutzer (Gast) können nur von anderen erstellte Quizze spielen.

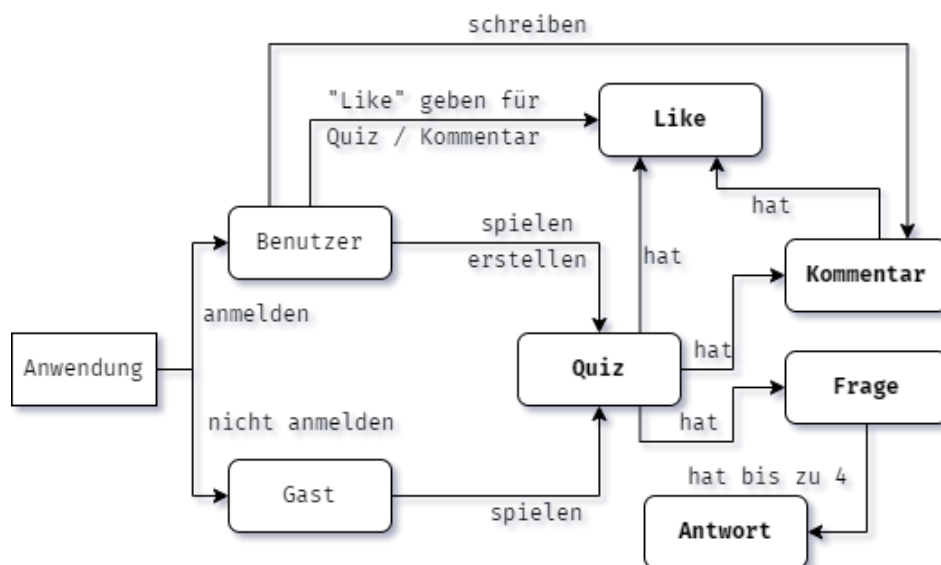
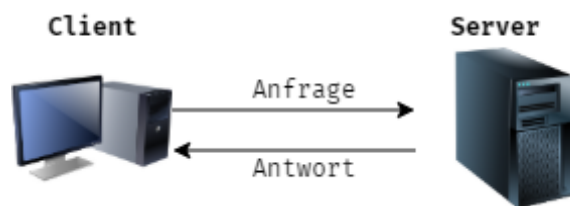


ABB. 1

## 2 | Grundlegendes

### 2.1 | Client-Server Modell



Das Client-Server-Modell ist ein Konzept, welche die Kommunikation zwischen Rechnern ermöglicht. Man unterscheidet zwischen dem (Web-)Server, welcher die Dienstleistungen bereitstellt und dem Client, der Anfragen (Request) an den Server stellt und eine Antwort (Response) von diesem erhält. ABB. 2

Der Client hat keine Kontrolle über den Server und ebenso andersherum. Hieraus folgt, dass z.B. die Validierung von Mindest-Passwortlängen nicht nur clientseitig (im Webbrowser) stattfinden darf, sondern auch immer serverseitig zu überprüfen ist. Da der Client clientseitig implementierte Sicherheitsmaßnahmen manipulieren kann beispielsweise durch CRSF-Angriffe<sup>1</sup>. Daher gilt folgender Grundsatz:

Der Server sollte einem Client niemals vertrauen.

Ein typischer Anwendungsfall ist folgender: Ein Benutzer ruft eine Webseite mit seinem Webbrowser (Client) auf, er schickt eine Anfrage (Request) an den Webserver, welcher daraufhin die Seite generiert (evtl. benötigte Datenbankabfragen ausführt, Templates laden etc.) und anschließend wird dies als Antwort (Response) an den Client zurückgesendet.

Ein Nachteil dieses Aufbaus ist die Abhängigkeit des Clients vom Server. Viele gleichzeitige Anfragen von Clients etwa durch einen Denial-Of-Service Angriff können den Server kurzzeitig überlasten bzw. funktionsunfähig machen.

---

<sup>1</sup> Siehe 4.7.5: Cross-Site-Request-Forgery

## 2.2 | Design Patterns

Design Patterns sind allgemein bewährte Lösungen für häufig auftretende Probleme in der Softwareentwicklung. Nachfolgend werden ausgewählte Entwurfsmuster erläutert um die folgenden Kapitel besser nachvollziehen zu können.

### 2.2.1 | Model-View-Controller

Das MVC-Muster teilt die Software in drei Komponenten: das Modell (*model*), die Ansicht/Präsentation (*view*) und die Steuerung (*controller*). Da es verschiedene Implementierungen zum MVC-Muster gibt, wird folglich die Realisierung in Laravel dargestellt.

Das Model stellt die Daten zur Verfügung und ist auch für das Erstellen, Ändern und Löschen dieser Daten verantwortlich. In Laravel basiert das Model auf einer *Active Record*<sup>2</sup> Implementierung, sodass ein Model zu einer bestimmten Tabelle in der Datenbank gehört. Nur das Model interagiert direkt mit der Datenbank.

Die View zeigt die Daten an, welche der Controller übergeben hat (oder nicht). Sie hat keinen Einfluss auf Model und Controller. Eine im Browser aufgerufene (Unter-)Seite wird dabei durch genau eine View dargestellt, welche aber auch aus mehreren Views bestehen (Templates) kann. Die View ist der einzig für den Benutzer sichtbare Teil der Webseite.

Der Controller ist die Schnittstelle zwischen Modell, View und dem Benutzer. Benutzereingaben werden an den Controller weitergeleitet, welche dieser validiert und gegebenenfalls benötigte Abfragen an das Modell ausführt. Das Ergebnis wird abschließend an die View weitergeleitet und dem Benutzer zurückgesendet.

---

<sup>[2]</sup> Siehe 2.2.3

## 2.2.2 | Dependency Injection

Dependency Injection (DI) ist ein Entwurfsmuster welches in der objektorientierten Programmierung eingesetzt wird um Abhängigkeiten von Objekten zu kontrollieren. Die Abhängigkeiten werden dabei durch den Konstruktor oder Setter-Methoden übergeben. Einer der Vorteile ist der einfache Austausch von Objekten durch andere. Nachfolgend ein Beispiel zur Veranschaulichung.

```
class Arbeiter
{
    private $hammer;

    public function __construct()
    {
        $this->hammer = new Hammer();
    }

    public function arbeite(){ ... }
}
```

ABB. 3

Ein Arbeiter benötigt ein Werkzeug zum Arbeiten. Ohne DI wird das Werkzeug (Hammer) direkt im Konstruktor erzeugt. Dies hat den Nachteil das ein Austausch des Hammers gegen ein anderes Werkzeug von außen nicht möglich ist, da der Arbeiter das Werkzeug lokal im Konstruktor erzeugt. Würde man jetzt ein anderes Werkzeug zuweisen wollen, wäre dies nicht möglich ohne Vererbungen bzw. Methodenüberschreibung.

Dem Arbeiter sollte außerdem egal sein, welches Werkzeug er verwendet.

Mithilfe Dependency Injection und einem Interface lässt sich das Problem lösen. Hierbei implementieren die Werkzeuge ein gemeinsames Interface, welches der Arbeiter annimmt.

Der Arbeiter hat nun keine Abhängigkeiten zu konkreten Klassen mehr und muss diese auch nicht kennen.

Stattdessen wird zur Laufzeit die benötigte Abhängigkeit welches ein bekanntes Interface implementiert, übergeben. Hierdurch lässt sich jederzeit

```
interface WerkzeugInterface
{
    function benutzen();
}

class Axt implements WerkzeugInterface{ ... }
class Hammer implements WerkzeugInterface{ ... }

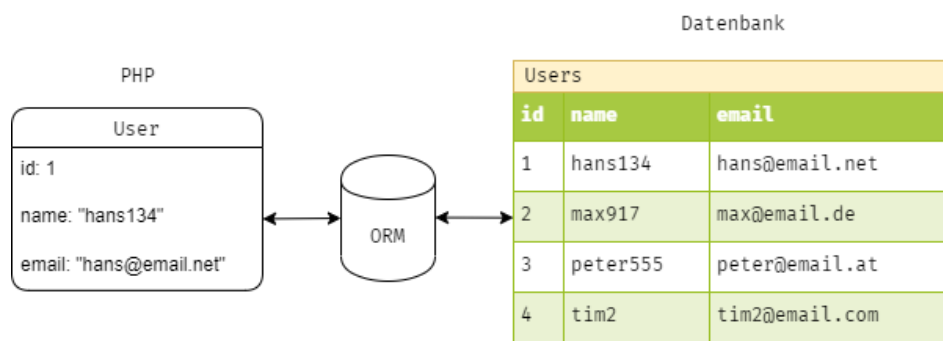
class Arbeiter
{
    public function arbeite(WerkzeugInterface $werkzeug)
    {
        $werkzeug->benutzen();
    }
}

$arbeiter = new Arbeiter();
$arbeiter->arbeite(new Axt());
$arbeiter->arbeite(new Hammer());
```

ABB. 4

das passende Werkzeug übergeben. Dependency Injection stellt einen wichtigen Bestandteil des Frameworks dar, welcher später noch näher erläutert wird.

### 2.2.3 | Active Record



Active Record beschreibt ein Muster welches ermöglicht Objekte aus einer Programmiersprache in einer relationalen Datenbank abzuspeichern („Objektrelationale Abbildung“ kurz ORM). Zu einer Tabelle in der Datenbank gehört dementsprechend eine eigene Klasse. Ein Objekt dieser Klasse stellt einen Datensatz (Zeile) in der Tabelle dar. Dieser Datensatz enthält alle Attribute aus der Tabelle als Objekt-Attribute.

Dieses Objekt kann in der jeweiligen Programmiersprache geändert und anschließend wieder in der Datenbank abgespeichert werden ohne dabei SQL-Befehle schreiben zu müssen. Dies hat den Vorteil das SQL-Injections und andere SQL Angriffe praktisch unmöglich werden. Und zudem wird der Code deutlich lesbarer.

Oft bieten die Klassen Methoden an um Objekte aus der Tabelle zu laden und diese zu modifizieren. Dabei werden auch eventuelle Fremdschlüssel ebenso zu einem passenden Objekt und man profitiert von den gleichen Möglichkeiten.

Intern wird dann eine dazugehörige SQL-Abfrage generiert und in der Datenbank ausgeführt.



## 2.3 | Technologie Stack

Im Folgenden werden die wichtigsten verwendeten Sprachen, Bibliotheken und Tools kurz zusammengefasst. Zunächst einmal der Unterschied zwischen Front- und Backend.


Frontend ist vereinfacht gesagt, der Teil einer Webseite mit dem der Nutzer interagieren kann bzw. der für ihn sichtbar ist. Dazu zählen unter anderem Buttons, Eingabefelder, Bilder etc. aber auch JavaScript-Dateien oder Stylesheets. Das Frontend zeigt die (meist) Daten aus dem Backend an.

Im Backend läuft die gesamte Verarbeitung der Daten ab, darunter zählen z.B. Datenbankabfragen oder E-Mails verschicken.

### Backend



➔ **PHP** („PHP: Hypertext Preprocessor“) ist eine serverseitige Programmiersprache welche in C von Rasmus Lerdorf 1994 für die Webentwicklung geschrieben wurde und unterstützt dementsprechend umfangreiche Funktionen wie z.B. Kommunikation mit Datenbanken.

PHP unterstützt sowohl eine prozedurale als auch eine vollständige objekt-orientierte Programmierung. Da PHP nicht vorher kompiliert werden muss können Datentypen in PHP dynamisch sein und müssen im Gegensatz zu kompilierten Sprachen wie z.B. Java nicht explizit deklariert werden. Objekt-orientierte Programmierung in PHP 7 hat starke Ähnlichkeit mit Java oder C#, außer einigen Syntaxunterschieden wie dem Pfeil (→) bei Methodenaufrufen anstatt einem Punkt (.). PHP wird auf 79%<sup>[3]</sup> aller Webseiten eingesetzt.



➔  Laravel ist ein objekt-orientiertes PHP-Framework geschrieben von *Taylor Otwell* in 2011. Es basiert auf dem Model-View-Controller Muster und verfügt über eine Vielzahl von nützlichen Funktionen wie eine Active Record Implementierung. Einzelne Framework-Komponenten werden noch separat erklärt.

---

<sup>3</sup> <https://w3techs.com/technologies/details/pl-php>

- ➔  Livewire<sup>4</sup> ist eine relativ neue Bibliothek (Erweiterung) für Laravel, welche eine einfache Kommunikation zwischen dem Front- und Backend erlaubt, indem Teile des Frontends direkt mit PHP/Laravel verknüpft & synchronisiert werden können. Normalerweise muss um Daten zwischen dem Front- und Backend auszutauschen eine API<sup>5</sup> im Backend programmiert werden um auf AJAX-Anfragen aus dem Frontend zu reagieren. Dieser Schritt wird hierdurch überflüssig, da dies intern von Livewire verwaltet wird.
- ➔  MySQL ist ein sehr beliebtes Datenbanksystem, welches unter anderem von YouTube, Facebook & Twitter, auch oft in Verbindung mit PHP verwendet wird. Vorteile sind u.a. die automatische Optimierung von SQL-Abfragen (insbesondere JOINS) und ein Zwischenspeicher (Cache).

### Frontend

- ➔  Bulma<sup>6</sup> ist das benutzte CSS-Framework in diesem Projekt und für das Aussehen der Webseite maßgeblich verantwortlich. Es beinhaltet verschiedene nützliche Elemente wie Buttons und vereinfacht & beschleunigt das Gestalten einer Seite.
- ➔ Turbolinks<sup>7</sup> beschleunigt die Webseite, indem beim Aufrufen einer Unterseite diese nicht komplett heruntergeladen und angezeigt werden muss, sondern nur der veränderte HTML-Body ausgetauscht wird und der Head-Teil (Skripte, Stylesheets etc.) bestehen bleibt. Hierdurch fühlt sich das Navigieren der Webseite für den Benutzer spürbar schneller & direkter an.
- ➔  Alpine<sup>8</sup> ist ein minimalistisches JavaScript Framework wodurch sich einzelne Elemente mithilfe von einfachen HTML-Markups interaktiv gestalten lässt, ohne dabei JavaScript separat schreiben zu müssen. Anwendung findet dies z.B. bei der Account-Einstellungen-Seite um zwischen den verschiedenen Tabs zu wechseln.

---

<sup>4</sup> <https://github.com/livewire/livewire>

<sup>5</sup> Application Programming Interface: Programmierschnittstelle

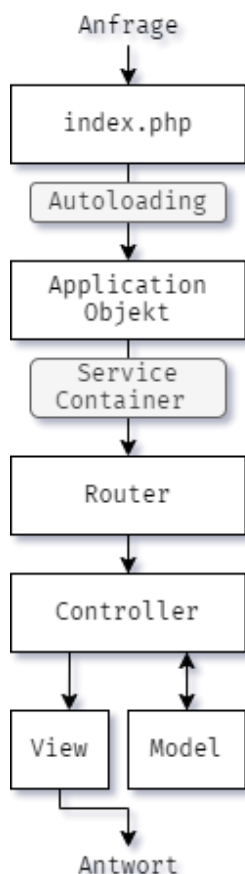
<sup>6</sup> <https://github.com/jgthms/bulma>

<sup>7</sup> <https://github.com/turbolinks/turbolinks>

<sup>8</sup> <https://github.com/alpinejs/alpine>

## 3 | Laravel

### 3.1 | Ablauf / Request Lifecycle



Um die folgende Beschreibung der wichtigsten Konzepte und Komponenten des Frameworks zu verstehen, erläutere ich kurz den Ablauf einer Anfrage eines Benutzers hin zur angezeigten Seite (auch „Request Lifecycle“). Folgendes Schema gilt für jede Anfrage, außer wenn es sich um statische Dateien wie Bilder, Stylesheets oder JavaScript handelt, diese werden nämlich direkt vom Webserver geliefert ohne Laravel zu durchlaufen.

Zunächst werden alle Anfragen eines Benutzers durch den Webserver (z.B. Apache oder NGINX) an die Datei /public/index.php weitergeleitet. Dort werden alle Klassen & Abhängigkeiten eingebunden, welches in PHP als „Autoloading“ bekannt ist, sodass diese später verwendet werden können. Dieser Schritt ist notwendig, da die Klassen nicht wie in Java vorher kompiliert werden müssen/können, sondern dynamisch eingebunden werden und PHP diese sonst nicht lokalisieren kann.

Als nächstes erstellt Laravel eine Instanz der Application Klasse, welches praktisch den Service Container darstellt.

Im nächsten Schritt wird die Anfrage durch den Router geschickt und falls eine passende Route existiert, wird der angegebene Controller aufgerufen. Andernfalls wird eine 404-Error-Seite generiert und zurückgesendet.

Im Controller wird die Anfrage validiert und gegebenenfalls benötigte Daten aus dem Model geladen.

Im letzten Schritt rendert der Controller die View mit diesen Daten aus dem Model/den Models und der Prozess beginnt von neu für die nächste Anfrage.

## 3.2 | Service Container

Der Service Container ist ein Dependency Injection Container (kurz: DIC) und beinhaltet eine Sammlung von verwendeten Objekten in Laravel, um Abhängigkeiten in Klassen zu steuern und Dependency Injection<sup>9</sup> auszuführen.

Hierdurch ist es möglich benötigte Objekte einfach in der Methodensignatur als Parameter markieren und Laravel übergibt automatisch dieses Objekt aus dem Service Container. Das Objekt muss vorher im Container hinterlegt worden sein.

Im folgenden Fall (Abb. 5) wird das Request Objekt, welches alle Daten der aktuellen Anfrage enthält, aus dem Container geladen und automatisch dem Konstruktor übergeben ohne sich um die Erstellung einer Instanz der Request-Klasse kümmern zu müssen.

```
class Test
{
    private Request $request;

    public function __construct(Request $request)
    {
        $this->request = $request;
    }
}
```

ABB. 5

Eine weitere Funktion des Service Containers ist die Verknüpfung eines Interfaces mit einer konkreten Klasse (Implementierung des Interfaces), sodass ein Interface als Methoden Parameter angegeben werden kann und dieses mit der Implementierung des Interfaces ersetzt wird.

---

<sup>9</sup> Siehe 2.2.2

### 3.3 | Eloquent ORM

„Eloquent“ heißt die Laravel Implementierung des Active Record<sup>10</sup> Patterns. Eine Tabelle der Datenbank hat (normalerweise) auch ein dazugehöriges Eloquent-Model also eine Klasse. Dies erlaubt Daten in der Tabelle beliebig zu modifizieren ohne eigenhändig SQL-Abfragen ausführen zu müssen.

Durch statische Methoden des Models `$quiz = Quiz::find(100);`  
können Datensätze aus der Tabelle geladen. `$quiz->title = "Anderer Quiz Titel";`  
In diesem Fall wird ein Objekt des Quiz- `$quiz->save();`  
Models mit der ID 100 zurückgegeben. ABB. 6: BEISPIEL

Laravel nimmt standardmäßig `id` als Primärschlüssel an.

Den Tabellennamen für eine SQL-Abfrage leitet Laravel dabei aus dem Plural vom Modelnamen ab, so folgt aus dem Quiz Model die dazugehörige Tabelle `quizzes`. Dieser Mechanismus lässt sich natürlich auch überschreiben, falls eine andere Tabelle verwendet werden soll.

In der zweiten Zeile (Abb. 6) wird das Titel-Attribut des Datensatzes verändert. Betrachtet man dies im Code wird einem auffallen, dass das Titel-Attribut in der Quiz Klasse überhaupt nicht definiert ist. Stattdessen wird in PHP eine sogenannte Magische Methode („magic method“) aufgerufen. In diesem Beispiel die `__set(...)` Methode welche immer dann ausgeführt wird, falls einem nicht vorhandenen Attribut Daten zugewiesen wird. Durch dieses Feature müssen Attribute nicht explizit in der Klassendefinition definiert werden.

Eine weitere hilfreiche Funktion seitens Laravel sind Zeitstempel bei Erstellung eines Datensatzes (`created_at`-Attribut) bzw. bei der Aktualisierung von Attributen (`updated_at`-Attribut) welche automatisch verwaltet werden.

---

<sup>10</sup> Siehe 2.2.3

### 3.3.1 | Beziehungen

Oftmals haben Models Beziehungen untereinander, so hat zum Beispiel ein Benutzer viele Quizze. Eloquent bietet hierfür eine Vielzahl von Methoden an und unterstützt alle gängigen Beziehungen wie 1:1, 1:n oder n:m Beziehungen. Beziehungen werden als Methoden im jeweiligen Model definiert.

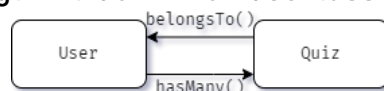
Möchte man nun eine 1:n Beziehung zwischen einem User und einem Quiz herstellen, so implementiert man eine Methode mit dem Namen der gewünschten anderen Klasse, in diesem Fall *quizzes* und ruft die *hasMany*-Methode mit dem Parameter `Quiz::class` auf (Abb. 7).

```
class User extends Model
{
    public function quizzes()
    {
        return $this->hasMany(related: Quiz::class);
    }
}
```

ABB. 7

Ist dieser Schritt erledigt, kann auf die Beziehung auf die gleiche Weise zugegriffen werden als wäre es ein Attribut. Beispielsweise um eine Liste der Quizze die vom Benutzer mit der ID 501 erstellt wurden zu erhalten könnte man dies durch einen simplen Methodenaufruf abfragen: `User::find(501)→quizzes`.

Intern wird die dazugehörige Tabelle des angegebenen Models gesucht (hier: „quizzes“-Tabelle) und eine WHERE-Bedingung angefügt mit dem Fremdschlüssel aus dem aktuellen Model bestehend aus Modelname + `_id` (also hier: `user_id` in „quizzes“). Die fertige



Abfrage sieht in etwa so aus: `SELECT * FROM quizzes WHERE user_id = 501`. Da es auch notwendig sein kann vom Quiz den dazugehörigen Ersteller zu finden kann man jede Beziehung umkehren, indem in gleicherweise wie oben beschrieben jetzt die *belongsTo*-Methode benutzt wird.

Manchmal kann ein Model zu verschiedenen anderen Models gehören, z.B. können Quizze und Kommentare „geliked“ werden. Hierfür teilen sich das Quiz- und Kommentar-Model das Like-Model, welches in der Datenbanktabelle in einer Tabelle („likes“) liegen. Ein daraus resultierender Vorteil ist die vereinfachte Abfrage von

Likes der Quizze und Kommentare. In Laravel wird dies als „*Polymorphic Relationship*“ bezeichnet.

### 3.3.2 | Soft Deletes

Manchmal ist es sinnvoll Einträge die vom Benutzer oder anderweitig gelöscht wurden nicht wirklich aus der Datenbank zu entfernen, sondern stattdessen intern als gelöscht zu markieren. Das hat den Vorteil Datensätze wiederherstellen zu können falls sie z.B. fälschlicherweise vom Benutzer gelöscht wurden.

Dies lässt sich relativ leicht umsetzen, da es bereits in Laravel implementiert ist. Um Soft Deletes zu benutzen, muss der Tabelle ein neues Datums-Attribut namens `deleted_at` hinzugefügt werden, welches den Zeitpunkt des Löschens angibt. Besitzt es den Wert `null` so wurde dieser Datensatz nicht gelöscht. Soft Deletes können für jedes Model einzeln aktiviert werden.

Theoretisch ist es auch möglich dies mit einem booleschen Attribut („`is_deleted`“) umzusetzen welches die gleiche Funktion hätte, allerdings dann ohne einen genauen Löschzeitpunkt.

Ab sofort wird bei sämtlichen Abfragen zusätzlich geprüft, dass das der Datensatz (noch) nicht gelöscht wurde (also `deleted_at = null`) um unerwünschte Inkonsistenzen zu vermeiden. Sollte aber auch bzw. nur nach gelöschten Datensätzen gesucht werden, so ist dies natürlich auch möglich mit dem Methodenaufruf von `withTrashed()` bzw. `onlyTrashed()` bei Abfragen. Um ein gelöschtes Objekt wiederherzustellen, so genügt die Ausführung der `restore()` Methode des Models, welches darauf das `deleted_at` Attribut in der Tabelle auf `null` zurücksetzt.

Möchte man letztendlich ein per Soft Delete gelöschtes Objekt endgültig löschen so bietet sich die `forceDelete()`-Methode dafür an, welche den Datensatz tatsächlich aus der Tabelle entfernt. Diese Aktion wird übrigens standardmäßig ausgeführt falls Soft Deletes für das Model nicht aktiviert wurden.

Dennoch besteht bei Soft Deletes unter Umständen ein großer Nachteil, denn da die Datensätze nicht wirklich aus der Datenbank entfernt werden bleiben die Primärschlüssel vergeben. Beispielsweise belegt ein mit Soft Delete gelöschter User den Benutzernamen, sodass dieser nicht wieder an neue Benutzer vergeben werden kann bis der Benutzer auch tatsächlich gelöscht wurde.

### 3.3.3 | Pagination

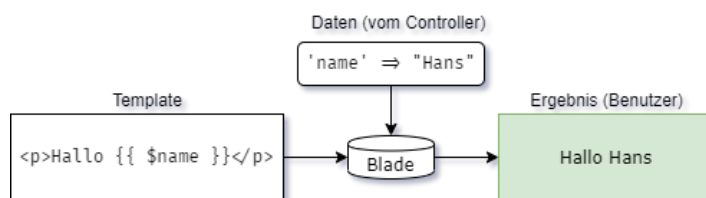


Oftmals arbeitet man mit größeren Datensammlungen welche beim Anzeigen nicht immer auf eine einzige Seite passen, daher bietet Laravel ein weiteres hilfreiches Feature welches Pagination oder auch Seitennummerierung genannt wird.

Durch den simplen Aufruf von `paginate()` lässt sich diese aktivieren. Um z.B. nur 15 Quizze pro Seite anzuzeigen, führt man `Quiz::paginate(15)` aus wodurch Laravel intern zur SQL-Abfrage eine LIMIT/OFFSET-Anweisung und in der URL die aktuelle Seitenzahl als Querystring (`?page=1`) anfügt.

Der LIMIT-Teil der SQL-Abfrage enthält die gewünschte Anzahl der Ergebnisse pro Seite und der OFFSET-Abschnitt ist ein Vielfaches von besagter Anzahl. Beispielsweise um die dritte Seite anzuzeigen, vorausgesetzt es sind genügend Datensätze vorhanden, wäre der angehängte SQL-Befehl `LIMIT 15 OFFSET 30`. Der Offset von 30 ergibt sich aus `Gewünschte Anzahl * (Seite - 1)`.

### 3.4 | Blade



Blade ist Laravels Template Engine. Eine Template Engine dient der Trennung vom Programmcode und Design zudem erlaubt es durch Vererbung von Templates die Wiederverwendung von Code.



Ein Template besteht aus regulärem HTML, CSS & JS Code und kann auch Platzhalter definieren, welche später durch die vom Controller übergebenen Daten ersetzt werden. Ebenso lassen sich Templates von anderen Templates vererben mittels `@extends(„anderes.template“)`, wodurch sich unnötige Wiederholungen durch Copy/Paste im Quellcode vermeiden lassen nach dem Motto „Dont Repeat Yourself“.

Selbstverständlich sind auch Kontrollstrukturen möglich wie `@if`, `@foreach` oder `@switch` um das Anzeigen von bestimmten Inhalten zu steuern.

Templates werden von Laravel beim Start einmalig eingelesen und zu einfachen PHP-Dateien kompiliert um keinen vermeidbaren Overhead und Verzögerungen zu verursachen, etwa durch das Parsen.

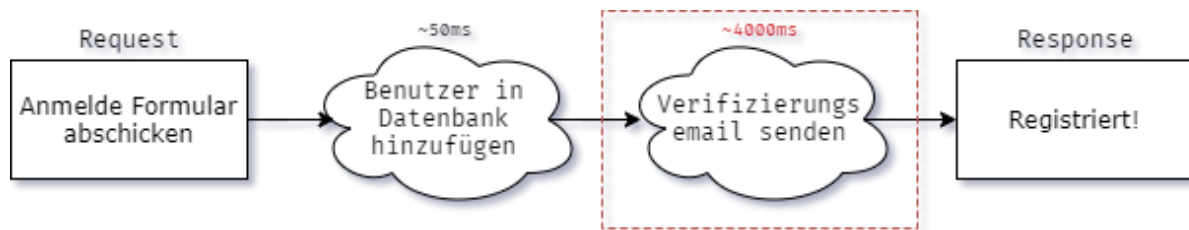
### 3.5 | Queue

Eine Queue / Warteschlange ist ein nach dem FIFO (First-In-First-Out) Prinzip funktionierende Datenstruktur. In Laravel werden in der Queue sogenannte Jobs verwaltet.

Ein Job kann dabei z.B. das Senden einer Verifizierungsemail, eine Benachrichtigung, dass jemand ein Quiz kommentiert hat, oder andere rechen- bzw. zeitintensive Operationen sein.

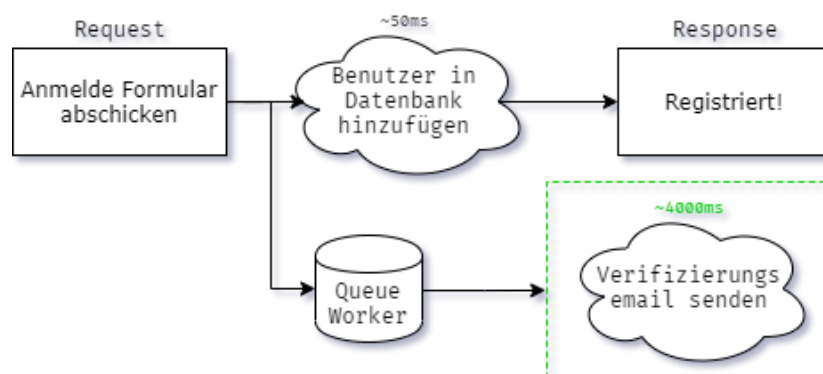
Der Queue Worker ist ein separater nebenläufiger Prozess der ununterbrochen läuft nachdem er gestartet wurde und eingehende Jobs in der Queue abarbeitet. Diese Nebenläufigkeit ist ein unverzichtbares Merkmal um Benutzeranfragen schnell verarbeiten zu können.

Ein Beispiel: Ein Benutzer möchte sich auf der Webseite registrieren. Nach dem Abschicken des Formulars wird der Benutzer der Datenbank hinzugefügt, welches noch relativ zügig verläuft. Folglich wird eine Verifizierungsemail an den Benutzer versendet doch aufgrund des normalerweise langsamen E-Mail-Versende Dienstes von mehreren Sekunden verzögert sich die Ausführung des Programms solange bis die E-Mail tatsächlich versendet wurde. Während dieser Zeit wartet der ungeduldige Benutzer vergeblich auf eine Antwort der Webseite. Für ihn erscheint die Seite als „eingefroren“.



Um dieses Problem zu beseitigen wird der Queue Worker eingesetzt. Der Ablauf ähnelt dem oben beschriebenen, nur das jetzt, anstatt die Verifizierungs-E-Mail direkt im gleichen Prozess zu versenden, es der Queue angefügt wird. Dadurch kann der eigentliche Prozess schneller durchlaufen werden, da er nicht mehr für das Versenden zuständig ist.

Die Queue, welche unabhängig vom restlichen Geschehen arbeitet, übernimmt nun die Aufgabe des Verschickens und der eigentliche Benutzer bekommt hiervon aufgrund der wegfallenden Verzögerung nichts mehr mit.



Nachdem der Queue Worker diesen Job erledigt hat ist er bereit für den nächsten Job.

## 4 | Quiz

### 4.1 | Vorüberlegungen



Zunächst wird zwischen einem bereits registrierten & angemeldeten Benutzer und einem unangemeldeten Nutzer (Gast) unterschieden werden. Gäste haben einzig allein die Möglichkeit bereits erstellte Quizze (von anderen Benutzern) zu spielen.

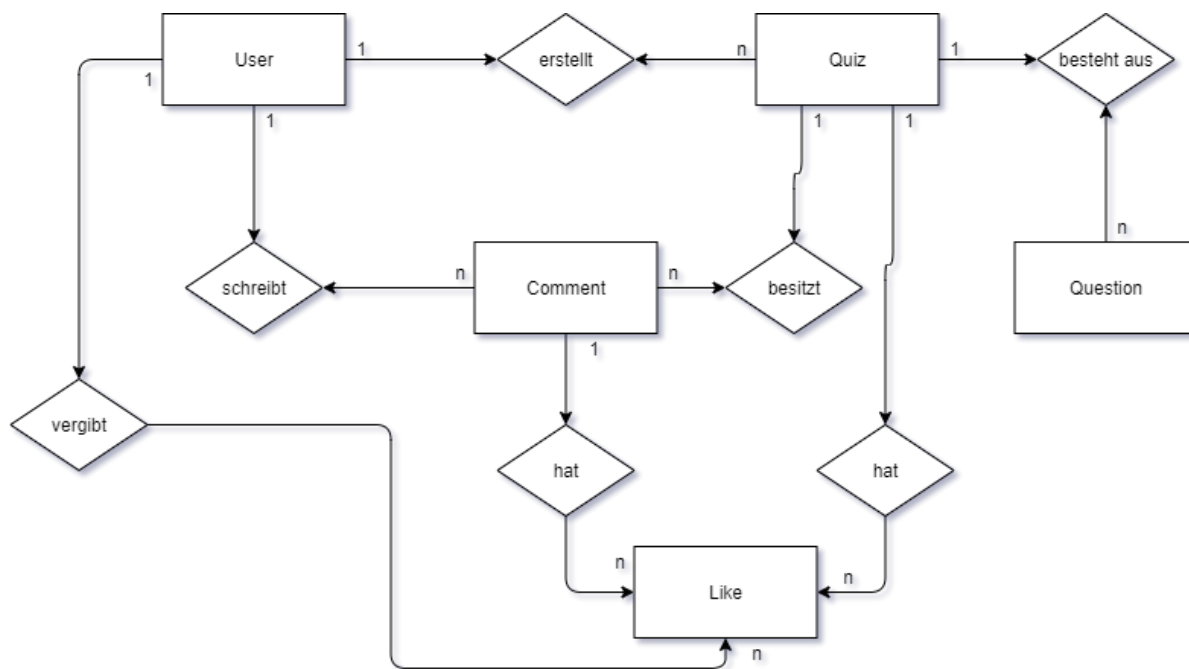
Ein angemeldeter Benutzer hingegen kann Quizze sowohl spielen als auch selber erstellen. Dabei kann er beliebig viele Fragen mit jeweils bis zu vier Antwortmöglichkeiten erstellen. Zu jeder Frage gehört genau eine richtige Antwort. Nach dem Erstellen kann das Quiz (außer Löschen) nicht mehr bearbeitet werden.

Jedes Quiz besitzt eine eigene Kommentarseite, auf der Benutzer kommentieren können. Ein Kommentar als auch ein Quiz kann von jedem Benutzer positiv bewertet werden. Bei neuen Kommentaren oder Likes soll der Quiz-Ersteller auf der Website benachrichtigt werden.

Jeder Benutzer muss seine E-Mail-Adresse nach dem Registrieren verifizieren um die Seite zu verwenden und kann anschließend auch den eigenen Account verwalten (E-Mail/Passwort/Benutzername ändern).

Auf der Startseite sollen Quizze angezeigt werden, welches jeder Besucher nach Datum, Beliebtheit (Meiste „Likes“) oder Aufrufen sortieren lassen kann. Zusätzlich wird es eine Suche geben um ein gewünschtes Quiz oder einen Benutzer schnell zu finden.

## 4.2 | Datenbank



Hinweis: Das obige ER-Diagramm ist nicht vollständig und zeigt nur relevante Entitätstypen aus Platzgründen.

Der Entitätstyp User besitzt neben den Attributen name (Benutzername) und email die Rolle des Benutzers (role), um zwischen regulären Benutzern und einem Administrator (role = 10) zu unterscheiden. Daneben befindet sich ein password-Attribut welches nicht das eigentliche Passwort, sondern nur dessen Hashwert<sup>11</sup> davon speichert und mehrere Zeitstempel für Soft Deletes, letztes Login-Datum & Datum der E-Mail-Verifizierung.

Der Entitätstyp Question hat einen Fremdschlüssel quiz\_id zur Quiz Tabelle, weil zwischen Question und Quiz eine 1:n Beziehung besteht. Da eine Frage aus bis zu vier Antwortmöglichkeiten bestehen kann werden, werden die Text-Attribute answer\_1 bis answer\_4 definiert. Zudem muss die richtige Antwort für diese Frage im correct Integer-Attribut angegeben werden. Die Fragestellung an sich findet sich im title-Attribut wieder.

Das Quiz selbst besitzt ein Fremdschlüssel zum User, um dem Quiz einem bestimmten Benutzer zuzuordnen. Des Weiteren werden Metadaten wie die Anzahl

<sup>11</sup> Siehe 4.8.4: Password Hashing

der Quiz-Aufrufe erfasst. Der Quiztitel und die Beschreibung sind jeweils im `title` bzw. `description` Attribut gespeichert.

Da jeder angemeldete Benutzer Kommentare zu einem bestimmten Quiz schreiben kann besitzt der Entitätstyp `Comment` gleich zwei Fremdschlüssel, einmal zum `User` und zum `Quiz`. Der Kommentarinhalt selbst steht dabei im `comment` Longtext-Attribut.

Der komplexeste Entitätstyp ist der `Like`-Typ, welcher zunächst die `user_id` als Fremdschlüssel zur `User`-Tabelle enthält um einen „Like“ auf einen Benutzer rückzuführen zu können. Da aber jetzt nicht nur ein Quiz, sondern auch Kommentare bewertet werden können müssen wir zwei (von Laravel vorgegebene) Attribute anfügen um Eloquent nutzen zu können.

Eines ist die `likeable_id`, welches den Primärschlüssel des Models enthält, der in der Spalte `likeable_type` angegeben wurde. Laravel wird beim Aufrufen dieser Beziehung, dann das richtige Eloquent-Model zurückgeben basierend auf der angegebenen ID und dem Typen. Hat beispielsweise der Benutzer mit der ID 133 das Quiz mit der ID 456 bewertet, dann werden der Tabelle *likes* diese Daten hinzugefügt.

...	id	user_id	likeable_id	likeable_type
...	1	133	456	App\Quiz

Möchte man z.B. das Objekt aus der Tabelle erhalten welches die ID von 1 hat, ist es möglich über das `likeable`-Attribut des Models das korrekte Eloquent-Objekt zu erhalten: `Like::find(1)→likeable` resultiert in einem Objekt vom Typ `Quiz` welches den Quiz-Datensatz mit der ID 456 repräsentiert.

## 4.3 | Router

Im Router werden die einzelnen URLs definiert, welche bestimmte Aktionen ausführen. Hauptaufgabe eines Routers ist die eingehende Anfrage eines Benutzers zum richtigen Controller weiterzuleiten oder falls keine passende Route existiert die 404-Fehler Seite anzuzeigen. Des Weiteren unterscheidet der Router zwischen den HTTP Methoden wie GET, POST, PUT, DELETE etc.

Nehmen wir folgende Definition aus dem Projekt als Beispiel, welches eine Quiz-Seite zum Spielen aufruft.

```
Route::get(uri: '/quiz/{quiz}/{slug}/play', [QuizController::class, 'show'])->name(name: 'quiz.show');
```

ABB. 8

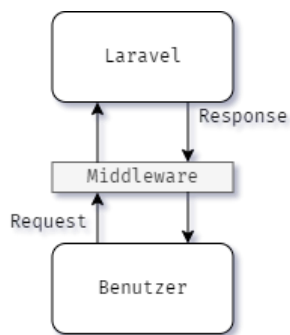
So wird hier eine eingehende GET-Anfrage mit der URL `/quiz/{quiz}/{slug}/play` dem `QuizController` zugewiesen und die dortige Methode `show(...)` aufgerufen. Um später in der View einfacher URLs generieren zu können, vergeben wir der Route noch den Namen `quiz.show`.

`{quiz}` und `{slug}` sind jeweils Platzhalter für eine Zeichenfolge und werden beim Methodenaufruf (`show`) als Parameter übergeben. Eine gültige URL die diese Route benutzen wird wäre z.B. `/quiz/100/tolles-quiz/play`. Die angegebene Methode würde nun das Quiz-Eloquent-Objekt mit der ID 100 als Parameter erhalten insofern in der Parameterliste das Quiz Model als Typ gesetzt wurde.

Dies funktioniert indem der Router die gegebene Controller-Methode durchsucht und falls der Parametertyp (z.B. `Quiz`) mit dem Platzhalter in der Routendefinition übereinstimmt (z.B. `{quiz}`). Existiert im betreffenden Model ein Datensatz mit dem Primärschlüssel aus dem Platzhalter, wird dieses Objekt der Methode übergeben.

Andernfalls wird eine 404 „Nicht gefunden“-Seite angezeigt. In Laravel nennt man dies Route Model Binding.

## 4.4 | Middleware



Middleware bieten die Möglichkeit ein- und ausgehende HTTP-Anfragen zu filtern bzw. zu modifizieren.

In Laravel werden diese verwendet um z.B. einen Benutzer auf eine Anmeldeseite weiterzuleiten, falls dieser nicht angemeldet ist oder CSRF-Tokens zu verifizieren.

Middlewares können einerseits global definiert werden damit alle Anfragen sie durchlaufen müssen oder auch pro Controller bzw. pro Controller-Methode für nur bestimmte Aktionen.

Beispielsweise soll der Benutzer angemeldet und über eine bestätigte E-Mail-Adresse verfügen bevor er Quizze erstellen, bearbeiten oder löschen kann. Einzig das Anzeigen (Spielen) des Quiz soll gestattet sein. Der einfachste Weg dies umzusetzen ist im Controller-Konstruktor die Middlewares zu registrieren:

```
$this->middleware(['auth', 'verified'])->except(['show']);
```

Demnach werden Anfragen an alle Methoden dieses Controllers, mit Ausnahme der `show()` Methode, zunächst die definierten Middlewares durchlaufen bevor sie an die jeweilige Methode weitergeleitet werden.

Ist der Benutzer in diesem Beispiel angemeldet & verifiziert wird die Middleware die Anfrage weitergeben an die nächste Middleware bzw. letztendlich dann dem Controller.

Man unterscheidet in Laravel zwischen „Before“- und „After“-Middleware. „Before“-Middlewares werden bei der Anfrage eines Benutzers noch bevor sie die Controller erreichen bearbeitet beispielsweise um die E-Mail-Bestätigung zu verifizieren. Erst danach werden sie dem Controller übergeben.

„After“-Middlewares werden ausgeführt nachdem eine Antwort an den Benutzer gesendet werden soll. Beispielsweise um die zu setzenden Cookies zu Verschlüssen.

## 4.5 | Controllers

Controllers stellen einen wichtigen Teil im Model-View-Controller Modell dar, denn sie kommunizieren mit Models um Daten zu erhalten und sie anschließend in einer View auszugeben.

Jedes Model besitzt einen dazugehörigen Controller der die Verwaltung des Modells übernimmt. Beispielsweise hat das Quiz-Model einen QuizController. Allgemein sollte ein Controller mindestens die vier grundlegenden CRUD Operationen (Create, Read, Update & Delete) verarbeiten können.

In Laravel stehen hierfür sogenannte „Resource Controllers“ zur Verfügung. Sobald diese in der Routendefinition aktiviert wurden, werden sieben einzelne Routen definiert, welche für die Aktionen index, create, store, show, edit, update & destroy stehen. Nachfolgend eine Übersicht.

Route	Aktion	Beschreibung
GET /quiz	index	Zeigt alle Quizze des Benutzers an. (in einer View)
GET /quiz/create	create	Zeigt dem Benutzer ein Formular (View) um ein neues Quiz zu erstellen. Das Abschicken löst die store-Aktion aus.
POST /quiz	store	Validiert die create-Anfrage und fügt der Datenbank ein neues Quiz hinzu.
GET /quiz/{quiz}	show	Zeigt das Quiz mit der ID {quiz} zum Spielen an. (View)
GET /quiz/{quiz}/edit	edit	Zeigt ein Formular (View) mit der das Quiz mit der ID {quiz} bearbeitet werden kann. Formular wird an die update-Aktion übermittelt.
PUT /quiz/{quiz}	update	Validiert die edit-Anfrage und aktualisiert den Datensatz mit der ID {quiz} in der Datenbank.

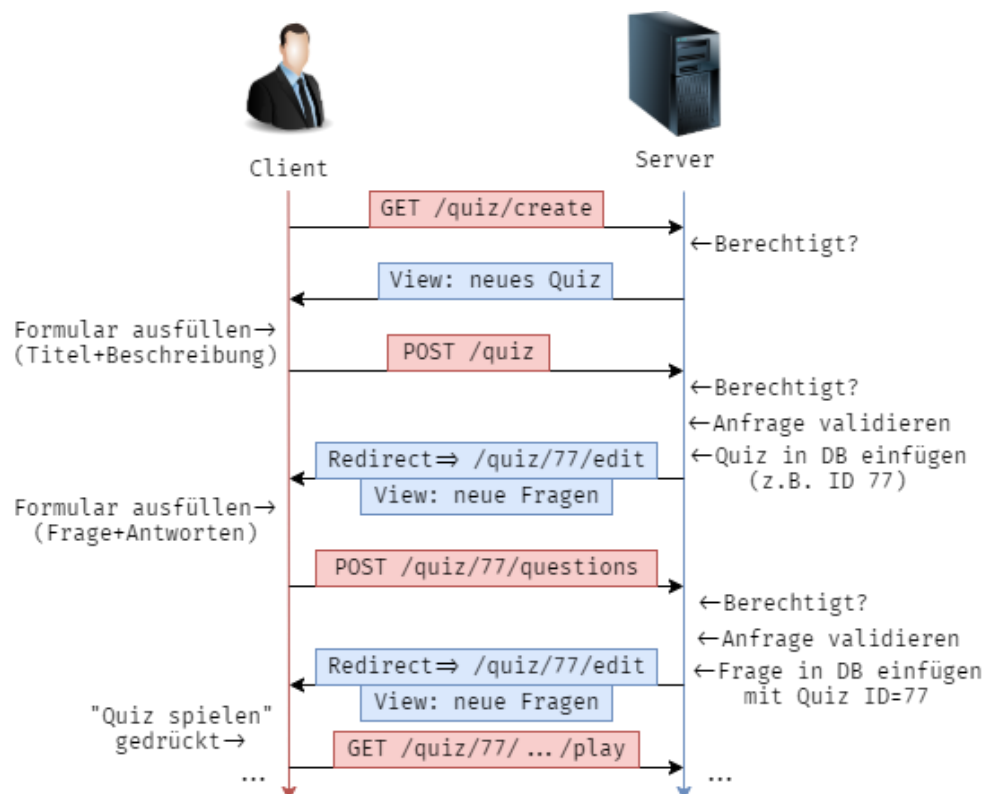


DELETE /quiz/{quiz}	destroy	Das Quiz mit der ID {quiz} wird gelöscht. Falls Soft-Deletes aktiviert sind, wird es soft deleted.
---------------------	---------	--

Jedes der oben dargestellten Aktionen sind Methoden des Resource Controllers.

Möchte man beispielsweise ein neues Quiz hinzufügen, dann ruft er die /quiz/create URL auf. Die Anfrage durchläuft zunächst die Middlewares, welche u.a. überprüfen ob der betreffende Nutzer angemeldet ist.

Insofern dies gegeben ist wird dem Benutzer eine View gerendert, welche ein Formular enthält um ein Quiz zu erstellen. Nachfolgend ein kurzes Ablaufdiagramm:



Das Abschicken des Formulars löst die store-Aktion im QuizController aus, welches daraufhin das neue Quiz erstellt und zu einer View weiterleitet auf der der Benutzer neue Fragen hinzufügen kann. Nach Senden dieses Formulars wird eine neue Frage zum Quiz hinzugefügt und zur gleichen Seite weitergeleitet um weitere Fragen zu erstellen.

## 4.6 | Models

Das Projekt verfügt über die 5 Models: Comment, Like, Question, Quiz & User. Diese Models verwalten die Daten und sind praktisch die Schnittstelle zwischen Controller und der Datenbank. Models enthalten neben den bereits erläuterten Beziehungen auch weitere Funktionen die wir uns anhand des Quiz-Models näher anschauen.

Ein Quiz gilt als spielbar/öffentlich, insofern für dieses mindestens eine Frage existiert. Möchte man nun alle Quizze abfragen, die auch Fragen besitzen wären sogenannte Query Scopes eine Option. Diese speziellen Methoden beginnen mit scope + einem Namen. Als Parameter wird der aktuelle Querybuilder übergeben, welchen wir dann beliebig verändern können. In diesem Fall wird eine WHERE EXISTS-Bedingung eingefügt um zu überprüfen, ob Einträge in der questions Tabelle vorhanden sind. Ist dieser Schritt getan lassen sich alle öffentlichen Quizze direkt abrufen ohne später zusätzlichem Code: `Quiz::public()→get()`.

```
public function scopePublic(BUILDER $query)
{
    return $query->has('relation: 'questions');
```

Arbeitet man unvorsichtig mit Eloquent-Beziehungen kann das N+1 Problem auftreten. Dieses besagt, dass wir bei jedem Zugriff auf die Beziehung eine zusätzliche Query ausführen müssen. Im obigen Beispiel bedeutet das, dass wir bei `$quiz→user` für jedes Quiz auf die user-Beziehung zugreifen und hierfür eine eigene Query gesendet wird: `SELECT * FROM users where id = {{user_id Fremdschlüssel in quizzes-Tabelle}}` plus dem eigentlichen `SELECT * FROM quizzes` Aufruf. Das wären bei 500 Quizzen 500 + 1 Abfrage und dementsprechend auch sehr lange Antwortzeiten.

```
$quizzes = Quiz::all();
foreach ($quizzes as $quiz) {
    echo $quiz->user->name;
}
```

Um diesem Problem entgegenzuwirken und die Anzahl auf konstante zwei Anfragen zu reduzieren, setzen wir auf Eager Loading. Dazu setzen wir bei dem Abrufen der Quizze die `with()`-Methode ein um alle user-Beziehungen direkt in einem Statement abzurufen: `Quiz::with('user')→all();` .

Die resultierenden zwei Queries wären einmal `SELECT * FROM quizzes` und `SELECT * FROM users WHERE id IN (1, 3, 6, 13, 14,...)`.

Ähnliches Beispiel mit `withCount`: Ein Quiz hat viele Bewertungen (Likes) von Benutzern. Wir möchten auf der Startseite die Anzahl dieser Likes ausgeben. Da wir sehr oft die Anzahl der Likes benötigen und nicht durchweg `withCount()` anhängen möchten fügen wir dem Quiz-Model das `$withCount = ['likes'];` Attribut hinzu. Jedes Model enthält daraufhin das neue Attribut `likes_count` mit den Anzahl der Likes, welches intern aus der `SELECT COUNT(*) FROM likes WHERE quiz...` Anweisung hervorgeht.

## 4.7 | Sicherheit

Bei der Webentwicklung gilt einiges zu beachten. Daher erläutere ich allgemeine Sicherheitsprobleme und deren Gegenmaßnahmen/Vorbeugung in Laravel.

### 4.7.1 | Validierung

Daten, welche der Benutzer z.B. über Formulare an die Webseite sendet sollten immer serverseitig validiert werden. Schaut man sich die `store`-Methode des `QuestionControllers` an, erkennt man das als Parameter ein `StoreQuestionRequest`, also eine Anfrage zum Speichern einer neuen Frage, akzeptiert wird.

Diese speziellen Anfragen sind `Form Requests`, in der eigene Regeln definiert und mit der eingehenden Anfrage abgeglichen werden. Laravel leitet automatisch Anfragen welche an diese Methode geroutet werden zuerst durch den `Form Request`. Sollte dieser feststellen, dass die Anfrage gegen die gesetzten Regeln verstößt, wird der Benutzer auf die Formularseite zurückgeschickt mit entsprechenden Fehlermeldungen.

Das `StoreQuestionRequest` definiert folgende Regeln:

```
"title" ⇒ "required|max:100",  
"answer_1" ⇒ "required|max:100",  
"answer_2" ⇒ "required_with:answer_3,answer_4|max:100",  
"answer_3" ⇒ "required_with:answer_4|max:100",  
"answer_4" ⇒ "max:100",  
"correct" ⇒ "integer|between:1,4",  
"order" ⇒ "integer|between:0,1000"
```

Einzelne Regeln werden mit einem senkrechten Strich (|) voneinander getrennt. Parameter beginnen nach einem Doppelpunkt und sind kommagetrennt.

Ein Titel/die Fragestellung muss immer gegeben sein (required) und darf die Maximallänge von 100 Zeichen nicht überschreiten (max:100). Eine 2. Antwortmöglichkeit ist optional außer es existiert eine 3. oder 4. Antwortmöglichkeit, denn dann ist sie erforderlich da ansonsten unerwartete Ereignisse auftreten könnten.

Wie bereits erwähnt werden bei ungültigen Anfragen automatisch Fehlermeldungen generiert, welche später in einer View mittels der `$errors` Variable angezeigt werden können.

#### 4.7.2 | Authentifizierung

Oftmals werden die Begriffe Authentifizierung und Autorisierung vermischt bzw. als Synonyme betrachtet, dies ist aber falsch.

Ein Benutzer gilt als authentifiziert, wenn er dem Server beweisen kann, dass er derjenige ist für den er sich ausgibt. In diesem Fall wird dies über ein Anmeldeformular realisiert indem die E-Mail-Adresse und das Passwort mit der Datenbank abgeglichen werden.

Autorisierung hingegen folgt nach der Authentifizierung und überprüft ob der Benutzer die Rechte hat eine Aktion auszuführen. Näheres im folgenden Abschnitt.

### 4.7.3 | Autorisierung

Die wichtige Bedeutung von Autorisierung zeigt sich besonders gut am folgenden Beispiel: Nehmen wir an, ein Benutzer hat nur ein Quiz mit der ID = 5 erstellt. Um es zu bearbeiten kann er zur URL `/quiz/5/edit` navigieren. Sollte es ein anderes Quiz geben z.B: ID = 4, könnte dieser Benutzer es auch durch den Aufruf von `/quiz/4/edit` bearbeiten, obwohl es ihm eigentlich nicht gehört.

Dies ist möglich da wir den Benutzer zwar als vollwertigen Benutzer authentifiziert haben aber nicht überprüfen ob dieser auch die entsprechenden Berechtigungen hat.

Laravel stellt hierfür sogenannte Policies bereit. Eine Policy gehört immer zu einem Model z.B. hat das Quiz Model eine QuizPolicy. In dieser werden Methoden definiert, welche das Quiz-Model und das aktuelle User-Model als Parameter erhalten. Für Unangemeldete wird immer **false** zurückgegeben, falls es nicht explizit definiert ist. Als Rückgabewert muss entweder ein **true** zurückgegeben werden, falls der Benutzer autorisiert ist oder andernfalls **false**, welches eine 401-Antwort dem Benutzer liefert. Insofern man Resource Controllers<sup>12</sup> verwendet, wird die Autorisierung durch vorgegebene Methoden erleichtert. Die sieben Methoden des Resource Controllers gehören verschiedenen Policy Methoden an:

Methode in...			
Controller	Policy	Erklärung der Berechtigung	Entscheidungsregel <sup>13</sup>
index	viewAny	Darf der (aktuelle) Benutzer eine Übersicht aller seiner Quizze ansehen?	Ja, immer
show	view	Darf der Benutzer ein bestimmtes Quiz ansehen?	

<sup>12</sup> Siehe 4.5

<sup>13</sup> Gilt nur für angemeldete Benutzer. Unangemeldete erhalten immer false.

create	create	Darf der Benutzer das Formular zum Erstellen eines neuen Quiz ansehen?	Ja, immer
store		Darf der Benutzer das Quiz (in der Datenbank) erstellen?	
edit	update	Darf der Benutzer das Formular zum Bearbeiten des gegebenen Quiz ansehen?	Ja, wenn die User_ID vom Quiz mit der aktuellen ID des angemeldeten Benutzers übereinstimmt.
update		Darf der Benutzer Änderungen am gegebenen Quiz (in der DB) speichern?	
destroy	delete	Darf der Benutzer das Quiz löschen? (Soft Delete <sup>14</sup> )	
Nicht standardmäßige, sondern eigene Berechtigungen			
forceDelete	forceDelete	Darf der Benutzer das Quiz dauerhaft löschen?	Ja, wenn die User_ID vom Quiz mit der aktuellen ID des angemeldeten Benutzers übereinstimmt.
restore	restore	Darf der Benutzer das Quiz widerherstellen	
/	like	Darf der Benutzer das Quiz liken?	Ja, wenn er es noch nicht bewertet hat.

In der View kann man dadurch mit den `@can(...)` bzw. `@cannot(...)` Anweisungen bestimmte Inhalte anzeigen oder verbergen je nach Berechtigung eines Benutzers.

---

<sup>14</sup> Siehe 3.3.2

#### 4.7.4 | Password Hashing

Unter Passwort Hashing versteht man das Hashen eines Passwortes um es anstatt eines Klartext-Passwortes sicher in der Datenbank abzuspeichern.

Dieser Schritt ist wichtig, da ansonsten im Falle eines unberechtigten Zugriffs auf die Datenbank alle Passwörter einfach entwendet werden und gegen diejenigen Nutzer verwendet werden könnten. Zudem werden Passwörter oftmals auch bei anderen Diensten verwendet, wodurch das ein großes Sicherheitsrisiko darstellt.

Hashing ist aber nicht mit Verschlüsselung gleichzusetzen. Verschlüsselte Daten können auch wieder entschlüsselt werden. Bei gehashten Daten ist dies unmöglich, vorausgesetzt es wird ein starker Hashalgorithmus wie bcrypt verwendet.

Laravel verwaltet das korrekte Hashen bei der Registrierung und Vergleichen des Passwortes beim Anmelden automatisch.

Trotzdem gilt zu beachten, dass dieser Aufwand des Hashens keinen Effekt hat, wenn das Benutzerpasswort zu schwach gewählt wurde, denn dann kann der Angreifer durch z.B. Brute-Force-Angriffe häufig verwendete Passwörter einfach beim Login ausprobieren und Zugriff erlangen.

#### 4.7.5 | Cross-Site-Request-Forgery (CSRF)

CSRF-Angriffe führen Aktionen im Namen eines Benutzers ohne dessen Kenntnis bzw. Einverständnis aus. Ein Angreifer könnte einen Benutzer auf eine schädliche Webseite weiterleiten, welche darauf Anfragen an unsere Webseite sendet. Da der Benutzer korrekt angemeldet ist, wird der Server die Anfrage normal bearbeiten und die geforderten Aktionen in seinem Namen ausführen.

Um dies zu unterbinden wird ein CSRF-Token bei jedem Formular hinzugefügt, welcher einen zufälligen String enthält. Ein Angreifer kann diesen String nicht erraten oder anderweitig herausfinden, zumal er sich bei jeder Nutzer-Session ändert.

## 4.8 | Notifications

Notifications bieten die Möglichkeit bei bestimmten Ereignissen (Events) Aktionen auszuführen z.B. eine Benachrichtigung dem Benutzer anzuzeigen. Diese Aktionen können über verschiedene Channels ausgeführt werden z.B. als E-Mail, Datenbank-Eintrag, Twitter-Nachricht, Pushmitteilung am Handy etc. In diesem Projekt werden ausschließlich E-Mail und Datenbank-Einträge verwendet.

Derzeit werden Benachrichtigungen gesendet, wenn sich ein neuer Benutzer registriert (für E-Mail-Bestätigung), ein Quiz kommentiert oder ein Quiz oder Kommentar bewertet wurde.

Datenbank-Benachrichtigungen werden im Falle des kommentierten Quiz an den Ersteller des Quiz gesendet, indem ein Eintrag in der notifications-Tabelle erstellt wird mit der Quiz-Ersteller-ID, Kommentator-ID und der Quiz-ID. Diese Tabelle neben diesen Daten auch eine Zeitstempel der dokumentiert wann eine Benachrichtigung abgerufen bzw. gelesen wurde.

In der View werden diese Daten dem betreffenden Benutzer angezeigt.

Bei der Mail-Benachrichtigung ist der Ablauf identisch, nur anstatt einem Datenbank-Eintrag wird eine E-Mail generiert.

Alle Benachrichtigungen werden vom Queue Worker<sup>15</sup> bearbeitet um schnelle Antwortzeiten zu garantieren.

---

<sup>15</sup> Siehe 3.5



## 4.9 | Front End

### 4.9.1 | Views

Views bestehen aus HTML-Code und zeigen die vom Controller übermittelten Daten an. Alle Views in diesem Projekt erben das Master-Layout, welches wiederum auch aus einzelnen Views besteht. Ein Controller kann eine View mithilfe der `view()`-Methode ausgeben. Neben dem Viewnamen können auch Daten mitgeliefert werden. Die View wird diese Daten dann an den Platzhaltern einsetzen.

Nachfolgend wird der verkürzte Aufbau der View, welche eine Liste der eigenen Quizze anzeigt, chronologisch dargelegt.

```
@extends('layouts.app')
```

Das aktuelle Layout erbt das Master Layout (`/layouts/app.blade.php`) und somit dessen HTML-Code.

```
@section('title', "Meine Quizze")
```

Das Masterlayout enthält ein Platzhalter-Feld für ein Titel. Dieses wird hiermit mit dem Inhalt „Meine Quizze“ überschrieben.

```
Quizze ({{ $user->quizzes->count() }}).
```

Befehle in `{{ }}` Klammern werden von Laravel/PHP verarbeitet & angezeigt. In diesem Fall wird die Anzahl der Quizze des Benutzers ausgegeben. Die Variable `$user` wurde vorher vom Controller übergeben.

```
<a href="{{ route('quiz.create') }}" class="button is-success">
```

Dieser Link leitet den Benutzer auf die Quiz-Erstellungs-Seite weiter. Der Name der Route stammt vom Resource-Controller `QuizController`, welcher die `create()`-Methode referenziert. Das Linkziel wäre in diesem Fall `/quiz/create`. Namen können bei Bedarf in der Routendefinition auch explizit geändert werden.

```
@forelse($quizzes as $quiz)
|   @include('layouts.quiz.edit', $quiz)
@empty
|   Nichts erstellt
@endforelse
```

Wie bereits erwähnt sind auch Kontrollstrukturen wie For-Schleifen möglich. Diese Forelse-Schleife ist eine spezielle Art der For-Schleife, welche bei leerem Eingabe-Array (hier: \$quizzes) das @empty Feld anzeigt.

---

```
{{ $quizzes→links() }}
```

Dieser Code Ausschnitt ist für die Seitennummerierung / Pagination zuständig. Er zeigt die Seitenzahlen und Weiter-/Zurück-Buttons an.

---

```
@if(!$user→quizzes()→onlyTrashed()→get()→isEmpty()) ... @endif
```

Falls der \$user gelöschte Quizze besitzt, wird der Körper der if-Bedingung ausgegeben. Andernfalls bleibt dieser verborgen und wird nicht mit zum Benutzer zurückgesendet.

Der noch fehlende wichtige Bestandteil einer View sind Formulare.

```
<form action="{{ route('quiz.like', $quiz) }}" method="post"
onsubmit="button.disabled = true;button.classList.add('is-loading')">
  @csrf
  @auth
    @can('like', $quiz)
      <button name="button" class="button is-danger grow">
        <i class="fas fa-heart m-r-sm"></i>Mag ich
      </button>
    @else
      <button name="button" class="button is-disabled is-light is-danger skew-forward"
        disabled>
        <i class="fas fa-check m-r-sm"></i>Mag ich
      </button>
    @endcannot
  @elseguest
    <p>Melde dich an um zu bewerten</p>
  @endauth
</form>
```

Dieses Formular stellt die Like-Funktion auf einer Quiz-Seite bereit. Das Ziel (action) dieser Form ist die Quiz-Like Route, welche die Signatur

POST /quiz/{id}/like besitzt. Würde diese Route nur eine PUT-Anfrage (etwa nach dem Bearbeiten vom Quiz) akzeptieren, müsste man dies in die Form mittels `@method('PUT')` setzen, da HTML-Forms nur GET und POST Anfragen unterstützen.

Der Platzhalter `id` wird Laravel durch das Attribut `id` aus dem Quiz-Model ersetzen. Ebenso wird die `@csrf` Markierung durch ein unsichtbares Eingabefeld mit dem entsprechenden CSRF-Token ersetzt.

Der Code zwischen `@auth` und `@elseguest` wird nur ausgeführt, insofern ein Benutzer authentifiziert<sup>17</sup> ist. Entspricht dies der Wahrheit wird überprüft ob der Benutzer autorisiert ist dem Quiz einen Like zu geben. Sollte auch das stimmen wird der innere Teil zwischen `@can` und `@else` ausgegeben, andernfalls der andere.

Teile der Webseite etwa bei der Account-Einstellungsseite verwendet die AlpineJS Library um die Webseite etwas interaktiver zu gestalten. Eine genaue Erläuterung würde aber den Rahmen dieser Arbeit sprengen daher verzichte ich drauf.

#### 4.9.2 | Livewire

Livewire ist eine Bibliothek für Laravel um die Kommunikation zwischen dem Backend und Frontend zu vereinfachen. Normalerweise ist das Senden von Benutzerformularen an den Server mit dem Neuladen einer Seite verbunden und zudem werden extra Routen benötigt. Diese Schritte fallen weg, da Livewire Benutzerformulare mit dem Backend synchronisieren kann.

Bei der Suche auf der Webseite wird Livewire eingesetzt, um sofort passende Quizze beim Eintippen anzuzeigen. Dies funktioniert indem die Eingabefelder mit gleichnamigen Attributen im Livewire-Backend per POST-Anfrage aktualisiert werden. In dieser Livewire-Klasse können diese Daten verarbeitet werden, etwa um die Datenbank nach passenden Quizzen zu durchsuchen. Anschließend wird eine kurze View gerendert und beim Benutzer (ohne Seitenreload) ausgetauscht.

Ähnlich läuft es bei der Quiz-spielen Seite ab.

---

<sup>16</sup> Siehe 4.7.5

<sup>17</sup> Siehe 4.7.2

## 5 | Fazit

Zwar konnte ich in dieser Arbeit nicht auf jedes Detail eingehen, dennoch glaube ich, dass dies einen kurzen Einblick in die komplexe Welt der Webseiten-Entwicklung mit Laravel geliefert hat.

## 6 | Literaturverzeichnis

- ➦ <https://laravel.com/docs/7.x> ⇒ Offizielle Laravel Dokumentation (v7.0)
- ➦ <https://github.com/laravel/framework>
- ➦ <https://www.microsoft.com/de-de/techwiese/know-how/was-ist-eigentlich-dependency-injection.aspx>
- ➦ <https://github.com/alpinejs/alpine>
- ➦ <https://laravel-livewire.com/>
- ➦ <https://bulma.io/>

---

© Alle Grafiken, sofern nicht anders angegeben, wurden von mir erstellt.

Verwendete Bildbearbeitungsprogramme: Inkscape & draw.io.

Verwendete Schriftarten: Bahnschrift (12) & Fira Code (11).