# Technische Dokumentation Music shuffle

Tatyana Vogel, Malte Dahlheim

2020-01-06

# Inhaltsverzeichnis

## Glossar

| Begriff | Bedeutung |
|---|---|
| wir | Wenn das Wort **wir** in dieser Dokumentation verwendet wird ist immer von Frau Vogel und Herr Dahlheim die Rede. |
| CSRF | Cross Site Request Forgery |
| Siler | PHP Library zum Erstellen von schlanken Webanwendungen |
| Twig | PHP Template Library |

## Projekt Planung

Die Planung des Projektes wurde mit der Hilfe von *Github Projects* durchgeführt. Dort haben wir Issues und Notes im Kanbanboard verfolgt.

### Arbeiten während des Projekts

Einzelne Funktionen wurden in eigenen Branches entwickelt und über *Pull Requests* in den Master Branch gemerged.

Das Mergen konnte vor der Endphase des Projekts nur von dem anderen Teammitglied durchgeführt werden. Damit wollten wir verhindern, dass ohne das Wissen das jeweils anderen Teammitglieds Änderungen am Projekt vorgenommen werden.

## Projekt Aufbau

### Framework

Die ursprüngliche Version des Projekts war als mit dem *SLIM Framework* geplant. Nach langem hin und er stellte sich aber heraus, dass das Projekt Skelett, welches wir verwendet hatte etwas zu viel Magie beinhaltet.

Nach kurzer Recherche entschieden wir uns das Projekt neu aufzusetzen, dieses Mal mithilfe der PHP Library Siler von Leo Cavalcante. *Siler* ist ein PHP Library die Stark von Prinzipien der funktionalen Programmierung inspiriert wurde. Etwas was besonders für *Siler* sprach, ist das die Library aus relativ

wenigen, leicht verständlichen Funktionen besteht. *Siler* macht keine Annahmen darüber wie die Funktionen eingesetzt werden.

Wir verwenden *Siler* hauptsächlich für das URL Routing aber hier und da nutzen wir ein paar der Helferfunktionen z. B. für den sichern Zugriff auf die PHP Superglobalen Variablen wie `$_SESSION`, `$_GET`, `$_POST`.

## Templating und CSS

Für Templates des Projektes verwenden wir Twig. Diese entscheid wurde getroffen, weil wir uns als Team einig waren, das PHP keine schöne Templating Sprache ist. Auch half uns diese Entscheidung, Logik und Views deutlicher von einander zu trennen.

Als CSS Framework verwenden wir das Utility Framework Tailwindcss. Tailwindcss erlaubt es uns, das wir uns beim erstellen der Templates verstärkt auf den Markup code fokussieren können und uns nicht über vermeintlich nützliche CSS Klassen Namen den Kopf zerbrechen müssen. Auch zwingt und Tailwindcss nicht in vorgegebene Markup Strukturen hinein wie es bei manchen anderen Mehrzweck CSS Frameworks der Fall ist.

## Datei Struktur

### Setup

Im Verzeichnis `bootstrap` wird das komplette Setup der Anwendung vorgenommen. Hier werden die Endpunkte registriert und Einstellungen geladen und die Twig Umgebung aufgesetzt.

### Eigene Logik

Dem Vorbild von *Siler* Folgend verwenden wir in unserem gesamten Projekt in Namespaces unterteilte Funktionen. Dies mag auf den ersten Blick ungewöhnlich erschienen, erwies sich in der Praxis aber genauso geeignet zum Organisieren des Programmcodes wie ein Standard OOP PHP Projekt, einzig Composer hat mit dem Autoimport von Funktionen in Namespaces noch Probleme.

Alle Businesslogik Funktionen befinden sich im Verzeichnis `funktions`

Insgesamt unterteilt sich unsere Anwendung in fünf Namespaces.

| Namespace | Anwendungsbereich |
|---|---|
| Auth | Funktionen für die Authentifizierung des/der Nutzer*In |

| Namespace | Anwendungsbereich |
|---|---|
| Database | Funktionen für das Arbeiten mit der Datenbank |
| Errors | Funktionen für das Verarbeiten von Fehlern |
| Validators | Funktionen für das Validieren von Eingaben |
| YouTubeAPI | Funktionen für das Arbeiten mit der YouTube API |

**Endpunkte**

Für wir verwenden im Projekt ein Datei basiertes routing. Jede Resource hat ein eigenes Unterverzeichnis im `endpoints` Verzeichnis, ausgenommen davon sind Endpunkte mit Session Bezug und der Homeendpunkt.

| Endpunkt | HTTP Verb | Server Action |
|---|---|---|
| / | GET | endpoints/home.php |
| /login | GET | endpoints/login.php |
| /login | POST | endpoints/auth_user.php |
| /logout | POST | endpoints/logout.php |
| /logout | GET | endpoints/logout.php |
| /register | GET | endpoints/register.php |
| /users | GET | endpoints/users/index.php |
| /users/create | GET | endpoints/users/create.php |
| /users | POST | endpoints/users/store.php |
| /users/{id} | GET | endpoints/users/show.php |
| /users/{id}/edit | GET | endpoints/users/edit.php |
| /users/{id} | PUT | endpoints/users/update.php |
| /playlists | GET | endpoints/playlists/index.php |
| /playlists/create | GET | endpoints/playlists/create.php |
| /playlists | POST | endpoints/playlists/store.php |
| /playlists/{id} | GET | endpoints/playlists/show.php |
| /playlists/{id}/play | GET | endpoints/playlists/play.php |

| Endpunkt | HTTP Verb | Server Action |
|---|---|---|
| `/playlists/{id}/edit` | GET | endpoints/playlists/edit.php |
| `/playlists/{id}/songs/create` | GET | endpoints/songs/create.php |
| `/playlists/{id}/songs` | POST | endpoints/songs/store.php |
| `/playlists/{id}/songs/{song_id}` | PUT | endpoints/songs/update.php |
| `/songs/{id}` | GET | endpoints/songs/show.php |
| `/playlists/{id}/edit` | GET | endpoints/playlists/edit.php |

### Codestyle

Im ganzen Projekt werden für von uns geschriebene Funktionen- und Variablennamen camelCase verwendet. Der Komplette PHP Code wurde mit **php-cs-fixer** überprüft und Formatiert.

## Überprüfung der Nutzer*Inneneingaben

Auf der Clientseite sind alle Eingabefelder, die Notwendig sind, mit dem required Attribute versehen. Für jedes Eingabefeld wird der entsprechende Input Type verwendet.

In dem Projekt *Music shuffle* werden alle Nutzereingaben überprüft, die dafür verwendeten Funktionen befinden sich im Namespace `Validators`.

Hier gibt es z. B. Funktionen wie `validPassword` welche ein Passwort String und einen Passwort Bestätigungsstring entgegennimmt und auf Richtigkeit überprüft. Die Überprüfung beinhaltet einen Check auf die Übereinstimmung der beiden Passwort Strings, einen Check, ob das Passwort lehr ist, sowie einem Check, ob das Passwort mindestens 8 Zeichen lang ist.

Insgesamt gibt es in unserer Anwendung 10 verschiedene Validator Funktionen:

`validPassword`, `validUsername`, `validEmail`, `validPlaylistId`, `validSongId`, `validUserId`, `validPlaylistname`, `validateYouTubeUrl`, `validateYouTubeId`, `validCSRFToken`.

Auf jeden einzelnen Validator genau einzugehen würde den Rahmen dieses Dokumentes sprengen. Die einzelnen Funktionen sind relativ kurz und einfach verständlich, daher empfehlen wir für näheres Verständnis einfach die Datei `validators.php` im Verzeichnis `funktions` durchzulesen.

Schlägt einer dieser Validator fehl, wird ein Fehler in der Session gesetzt und der/die Nutzer*In wird zu ihrem Ursprungsort redirected.

Für die Verwendung eines Validators muss einfach die Funktion aus dem Namespace importiert werden und kann dann wie jede andere Funktion aufgerufen werde.

```php
<?php

use function Validators\validUsername;

$username = validUsername($_POST['username']);
```

### Schreiben in die Datenbank (SQL Injections)

Nuerzer*Innen eingeben werden nur über prepared Statements in die Datenbank geschrieben. Für den Zugriff auf die Datenbank verwenden wir *PDO*. Durch die prepared Statements verhindern wir die Injection von SQL Statements durch böswillige Nutzer*Innen.

## Script Injection

Im ganzen Projekt wird darauf geachtet, dass es zu keinen Script Injections kommen kann. Eingaben wie die des Usernames werden mit `htmlspecialchars` escaped. Twig übernimmt für uns auch das escaping von Strings in den Views, somit haben wir auch einen gewissen Schutz, falls wir das Validieren einer Eingabe mal vergessen sollten.

## Session Highjacking

Nach jedem Login wird die Session ID mittels `session_regenerate_id()` neu generiert.

## CSRF Protection

Jedes Formular beinhaltet ein verstecktes Input Feld welches als Wert einen SHA1 Hash hat. Der Hash wird aus einem zufällig generierten 255 Bit langen String erstellt.

Dieser Hash wird ebenfalls in der Session des/der Nutzer*Inn gespeichert. Bei jeder abgeschickten Formular, welches die Methode POST verwendet wird der dieser Hash mit dem in der Session gespeicherten Hash verglichen. Wenn die beiden Hashes nicht übereinstimmen, wird ein Fehler angezeigt.

## Testing

Ursprünglich gab es den Gedanken unsere Anwendung mit End to End Test, voll automatisch zu testen. Aus Zeitgründen wurde das Projekt nur Manuel getestet.