Projektdokumentation

Modul

Web-Engineering II

Projektname

"Shelfy" – File Manager Web-App

Autor

Vigan

Matrikelnummer

**

Abgabedatum

22.06.2025

GitHub-Repository

https://github.com/vKekz/File-Manager

TECHNISCHE DOKUMENTATION

Beschreibung

Das Projekt “Shelfy” – File Manager wurde im Rahmen des Moduls Web Engineering II entwickelt und verfolgt das Ziel, eine moderne sowie sichere Webanwendung zur Verwaltung von Ordnern und Dateien bereitzustellen. Benutzer haben die Möglichkeit, sich zu registrieren, um dadurch einen eigenen Speicher zu erhalten. Darin können sie Ordner erstellen, Dateien hochladen und dementsprechend verwalten. Die Anwendung berücksichtigt grundlegende Aspekte der Webentwicklung, darunter saubere HTML-Struktur, responsives Design, serverseitige Logik mit PHP sowie die Anbindung an eine relationale MySQL-Datenbank.

Verwendete Technologien

XAMPP	Lokale Entwicklungs- und Testumgebung mit Apache und MySQL
MySQL	Relationales Datenbankverwaltungssystem
PHP 8.2	Serverseitige Verarbeitung der Rest-API Anfragen und Datenbankanbindung
Angular 19.2.11	Web-App Framework für UI/UX sowie Client-Verbindung für Rest API
HTML	Strukturierter Aufbau des User-Interface
CSS	Responsive Design über Stylesheets
TypeScript	Typisierte Programmiersprache aufgebaut auf JavaScript
PhpStorm	Code-Editor für PHP und Frontend
PostMan	HTTP-Client zum Testen von Anfragen

Allgemeine Projektstruktur

Zur Projektstruktur wurden sich sehr viele Gedanken gemacht. Bereits zu Beginn war es klar eine Abgrenzung zwischen **Backend** und **Frontend** zu schaffen. Es später wurde nochmal die eigentliche **Anwendung** vom **Kern** abgegrenzt, um genau zu wissen, was das Framework ausmacht. Hilfe wurde sich dabei über verschiedene Analysen von bereits bestehenden großen PHP-Anwendungen geschaffen (Vorgeschlagene Projektstruktur).

Die wichtigsten Verzeichnisse sind wie folgt gegliedert:

File-Manager/

```
│
├ Backend/                # PHP Rest-API
│  ├ config/              # App Konfiguration
│  ├ public/              # API Boot Entry Point
│  ├ src/
│  │  ├ App/              # App Logik (Business Layer)
│  │  └ Core/             # Model-View-Controller Framework
│  ├ vendor/              # Composer Abhängigkeiten
│  ├ .env                 # Umgebungsvariablen (Produktion)
│  ├ .env.dist            # Vorlage für Umgebungsvariablen
│  └ composer.json        # Composer Definitionen
│
├ Frontend/               # Frontend (Angular Projekt)
│ ├ src/
│  ├ app/
│  └ components/          # UI-Komponenten (Pages & Elemente)
│  ├ guards/                    # Angular Guards
│  ├ services/            # Services zur Client Anbindung für Anfragen an Rest-API
│  └ states/              # State-Verwaltung mit In-Memory-Cache
```

Backend

Allgemein dient das Backend als REST API, die über eine angegebene **/api** URL angesprochen werden kann. Das bedeutet, dass mit der Anwendung über alle HTTP-Methoden wie GET, POST, PATCH und DELETE kommuniziert werden kann.

REST API

Wie bereits erwähnt dient das Backend als REST API, eine einfache Schnittstelle, die das Client-Server-Modell repräsentiert.

Client (Angular)

↓

HTTP-Anfrage (Route, Methode, Payload, Access Token)

↓

Backend (PHP)

↓

Router → Middleware → Controller → Service → Repository (Datenbank)

↓

HTTP-Antwort (JSON)

↓

Frontend (UI, State)

Um das ganze so generisch wie möglich zu gestalten, wurde sich dafür das **Model-View-Controller** Pattern angeschaut und angewendet, jedoch auf das Projekt bezogen leicht angepasst.

Model-View-Controller

Jedes Mal, wenn unsere Anwendung eine Anfrage über das Frontend also Angular implementierte UI erhält, wird diese am App-Entry-Point in kleine “Paket-Informationen” gespalten. Dazu gehört die Route, die HTTP-Methode sowie alle mitgesendeten Daten wie Parameter.

Anhand der Route kann die Anwendung entscheiden, welcher der registrierten Controller und dementsprechende Methode mit den mitgesendeten Parametern aufgerufen werden soll und führt dies dann aus. Um nicht jede neue Controller-Methode jedes Mal einzeln zu registrieren wurde dieser Schritt vereinfacht über Attribute gelöst. Das bedeutet es gibt eine Liste an Attributen, die eine jeweilige Http-Methode repräsentieren (z.B.: “HttpGet.php” für GET-Anfragen) welche an eine beliebige Controller-Methode angehängt werden können. Die Anwendung kann dann mit Hilfe von Reflection alle mit einem Http-Attribut registrierten Methoden auslesen und für den jeweiligen Controller und die jeweilige Route in ein Array speichern.

Sollte die Route ungültig sein und kein Controller oder keine Methode gefunden werden wird ein Method Not Allowed also ein HTTP-Code 405 zurückgegeben.

Hier ist eine Liste der Controller und deren Typen der Verarbeitung:

1. Auth: Anfragen zur Authentifizierung (Registrierung und Anmeldung)
2. User: Anfragen für Benutzer (Einstellungsänderungen und Abmeldung)
3. Session: Anfragen zu Sitzungen (Validierung)
4. File: Anfragen zu Dateien (Hochladen, Löschen und Herunterladen)
5. Directory: Anfragen zu Ordnern (Erstellen und Löschen)
6. Storage: Anfragen zum Speicher (Speicher finden oder durchsuchen)

Jeder Controller dient allgemein nur als Weiterleitung der Anfragen, die Verarbeitung findet dann im repräsentativen Service statt, der vom jeweiligen Controller aufgerufen wird. Dieser beinhaltet die eigentliche Business-Logik sowie Datenbankanbindung.

Um weiterhin eine saubere Struktur zwischen allen Services zu halten, d.h. sich nicht um manuelle Instanziierung von Klassen zu kümmern wurde eine angepasste Dependency-Injection implementiert.

Dependency-Injection

Früher kannte man es als Instanziierung einer Klasse, welche dann in einem jeweiligen Konstruktor einer anderen Klasse wiederverwendet wurde, falls gebraucht. Heutzutage kümmert sich genau dafür die Dependency-Injection.

Am App-Entry-Point, noch bevor die Anfrage bearbeitet wird, werden alle möglichen Handler, Services sowie Controller als Singleton im Service-Container registriert. Singleton bedeutet, dass im ganzen Anfragen-Zyklus für die jeweilige Klasse nur eine Instanz existiert.

Je nachdem wo welche Klasse bzw. Service gebraucht wird, kann dieser ganz bequem über den Klassen-Konstruktor (Beispiel) “injected”, also automatisch von der Dependency-Injection instanziiert werden.

Datenbank

Entitäten

Die Anwendung basiert auf einem relationalen Datenbankmodell mit vier zentralen Tabellen:

1. **user_entities**: Speichert alle registrierten Benutzer
    a. Id (varchar, Primärschlüssel)
    b. UserName (varchar)
    c. Email (varchar)
    d. EmailHash (varchar)
    e. PasswordHash (varchar)
    f. PrivateKey (varchar)
    g. CreatedAt (datetime)
    h. Settings (json)
2. **session_entities**:
    a. Id (varchar, Primärschlüssel)
    b. UserId (varchar, Fremdschlüssel zu user_entities(Id))
    c. DeviceDate (varchar)
    d. IssuedAt (datetime)
    e. ExpiresAt (datetime)
3. **directory_entities**:
    a. Id (varchar, Primärschlüssel)
    b. ParentId (varchar, Fremdschlüssel zu directory_entites(Id))
    c. UserId (varchar, Fremdschlüssel zu user_entities(Id))
    d. Name (varchar)
    e. NameHash (varchar)
    f. Path (varchar)
    g. CreatedAt (datetime)
    h. IsRoot (boolean)
4. **file_entities**:
    a. Id (varchar, Primärschlüssel)
    b. DirectoryId (varchar, Fremdschlüssel zu directory_entities(Id))
    c. UserId (varchar, Fremdschlüssel zu user_entities(Id))

d. Name (varchar)
e. NameHash (varchar)
f. RealHash (varchar)
g. Hash (varchar)
h. Size (int)
i. UploadedAt (datetime)

Beziehungen

Die Beziehung zwischen user_entities und allen anderen Tabellen ist 1:n, d.h. ein Benutzer kann z.B. mehrere Sitzungen, Dateien und Ordner haben.

Die Beziehung zwischen directory_entities und directory_entities ist auch 1:n, d.h. ein Ordner kann mehrere Ordner haben.

Die Beziehung zwischen directory_entities und file_entities ist auch 1:n, d.h. ein Ordner kann mehrere Dateien haben.

Verschlüsselung

Shelfy der File – Manager hat das Ziel sicher zu sein, dazu gehört auch unter anderem eine robuste Verschlüsselung. Für die Verschlüsselung wurden sich auch viele Gedanken gemacht. Es wurde viel experimentiert doch am Ende blieb es bei einer symmetrischen AES-256-GCM Verschlüsselung mit privatem Schlüssel, der zunächst für alle globalen Verschlüsselungen gilt.

Als weitere Sicherheitsmaßnahme werden Benutzer spezifische Daten mit einem pro-Benutzer generierten Schlüssel verschlüsselt. Dieser wird einmalig bei Registrierung erstellt.

Hier ist eine Liste der wichtigsten Daten, die pro-Benutzer verschlüsselt werden:

- Benutzer-Daten wie Name, E-Mail und Passwort
    - Passwörter werden vorher zusätzlich mit dem Standard Argon2Id über password_hash() gehasht
- Ordnernamen und -pfade
- Dateien
    - Namen
    - Hier war es auch wichtig, dass der Inhalt von hochgeladenen Dateien verschlüsselt wird

Jetzt stellt man sich wahrscheinlich schon die Frage, wie das mit dem Suchen in der Datenbank funktioniert, denn schließlich werden alle Daten beim Verschlüsseln anders verschlüsselt, d.h. ein Input führt immer zu einem anderen Output aufgrund von zufälligem Initialisierungs-Vektor. Und das könnte z.B. problematisch werden beim Suchen von E-Mails bei der Benutzeranmeldung. Den wie sucht man nach einem zufälligen Wert?

Dazu kam die Idee nicht nur die verschlüsselte E-Mail zu speichern, sondern auch eine mit Privatem Schlüssel gehashten Signatur der E-Mail. Anstatt nun nach der verschlüsselten E-Mail in der Datenbank zu suchen, wird nach der privaten Signatur der E-Mail gesucht, die wiederum deterministisch also jedes Mal gleich ist. Das gleiche Prinzip wird für Ordner- und Dateinamen verwendet.

Entschlüsselt werden nur die Daten, die zu jeder Anfrage gebraucht werden. Auch beim Dateiinhalt war es wichtig, dass wenn eine Datei heruntergeladen wird, der entschlüsselte Inhalt nur In-Memory über php://output zum Benutzer gestreamt wird. Das bedeutet, dass auf dem Dateisystem niemals eine Kopie der entschlüsselten Datei erstellt wird.

Damit nicht jeder Zugriff auf die REST API hat und kein Benutzer auf Dateien eines anderen Benutzers zugreifen kann muss sich jeder pro Anfrage authentifizieren.

Authentifizierung

Die Authentifizierung wurde über ein angepasstes JWT (JSON-Web-Token) Handling implementiert. Jedes Mal, wenn eine neue Sitzung erstellt wird, also beim Registrieren oder Anmelden eines Benutzers, wird ein Access Token generiert, welcher nur zu dieser Sitzung und diesem Benutzer passt. Dieser Access Token beinhaltet Informationen über den Benutzer, die Sitzung, wann er generiert wurde und wann er abläuft.

Beispiel Anfang eines Access Tokens: “eyJhbGciOiJzaGE1MTIiLCJ0eXA…”

Damit dieser nicht verloren geht, wird er im Local-Storage des Browsers so lange aufbewahrt, bis er nicht mehr valide ist.

Wie bereits erwähnt darf nicht einfach jeder Zugriff auf die REST API haben, dazu prüft die REST API erstmal, ob die Route also der Controller für jeden zugänglich ist, dies wird über das Authorize-Attribute festgelegt. Falls dies nicht der Fall sein sollte und die Route nur über Authentifizierung erreichbar ist, muss einer der generierten Access Tokens des Benutzers als Request Header an die Anfrage mitgesendet werden. Standardmäßig wird dazu der Authorization Header benutzt.

Da aber nicht jeder Benutzer das manuell erledigen möchte kümmert sich ein sogenannter Angular HTTP-Interceptor, welcher jede client-seitige Anfrage vorher abfängt, überprüft, ob diese zur REST API gesendet wird und ein Access Token im Local-Storage vorhanden ist. Stimmt alles überein, so wird der jeweilige Access Token automatisch als Authorization Header mitgesendet.

Die REST API kann nun endlich den Access Token validieren. Das passiert indem überprüft wird, dass dieser nicht manipuliert worden ist (Hash-Signatur), nicht abgelaufen ist, ein tatsächlicher Benutzer und eine tatsächliche Sitzung zum Access Token existieren und die Benutzer-IDs übereinstimmen. Erst dann kann sich die REST API sicher sein, die Anfrage zu bearbeiten und dem Benutzer eine Antwort zu liefern. Während des Überprüfungsprozesses werden wichtige Daten zum Benutzer im Kontext der Anfrage gespeichert, um sie für zukünftige Operationen bereit zu haben.

Sollte ein Access Token nicht validiert werden können, d.h. er ist zum Beispiel abgelaufen, wird die Anfrage mit einer Unauthorized Antwort also HTTP-Code 401 vorher unterbrochen. Keine sensiblen Daten werden unaufgefordert zurückgegeben.

Aus Sicherheitsgründen laufen Sitzungen in Shelfy automatisch standardmäßig nach 5 Stunden ab, danach muss sich der Benutzer neu anmelden.

Konfiguration

Für die Konfiguration gibt es zwei allgemeine Dateien. Die tatsächliche Konfigurations-Datei mit folgenden Einstellungen:

- “frontendUrl”: URL zum Angular Frontend
    - Wird auch benutzt, um die CORS-Header festzulegen
- “environmentFile": Dateiname der Umgebungsvariablen
- “authenticationTokenLifetime”: Lebensdauer eines Access Tokens

Und die eben erwähnten Umgebungsvariablen, die nicht jeder öffentlich sehen sollte:

Datenbankspezifisch:

- “DB_HOST”: Hostadresse
- “DB_USER”: Benutzername
- “DB_PASSWORD”: Passwort
- “DB_NAME”: Name der Datenbank

Kryptografisch:

- “HASH_MASTER_KEY”: Globaler Schlüssel für Signaturen
- “ENCRYPTION_MASTER_KEY”: Globaler Verschlüsselungsschlüssel

Beides wird über Configuration.php und Environment.php verwalten.

Frontend

Das Frontend wurde mit Hilfe des Angular Framework geschrieben. Angular ist komponentenbasiert, das heißt alles, was man im UI sieht, kann in mehrere Komponenten aufgeteilt werden. Sei es eine ganze Seite oder ein Button. Diese Komponenten bestehen wiederum aus drei Dateien:

- CSS-Datei: Beinhaltet Style der Komponente
- TypeScript-Datei: Beinhaltet client-seitige Logik
- HTML-Datei: Beinhaltet Struktur sowie Verknüpfung zur TypeScript-Datei mit Hilfe von Angular

Hier ist eine kurze Liste an Komponenten in Shelfy:

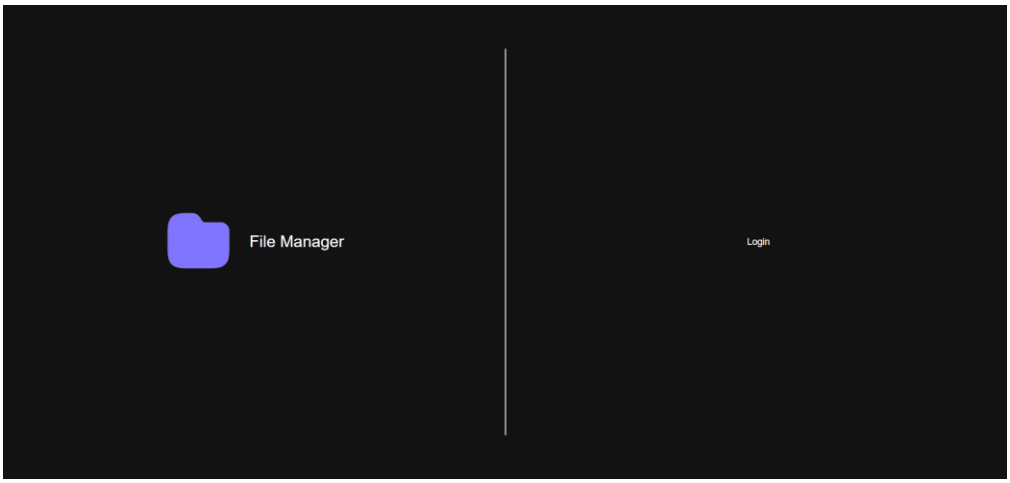
- DropDownMenu: Drop-Down-Menü für Ordner und Dateien
- Home: Landing Page für Shelfy
- MainBottomNav: Responsive Navigation im unteren Bereich der Webanwendung
- StorageTable: Tabelle die die Ordner und Dateien anzeigt
- und viele mehr…

Wie bereits erwähnt kann Angular auch HTTP-Anfragen senden, diese werden für jeden API-Controller repräsentativ über eigene Services im Frontend versendet. HTTP-Anfragen können entweder direkt über diesen Service oder über einen State erstellt werden. In Shelfy ist ein State eine Art In-Memory-Speicher, der die erhaltenen Daten wie eine Liste an Ordner verarbeiten und für das UI speichern kann. Es ist schließlich wichtig, dass das UI flüssig und direkt aktualisiert wird, man sollte als Benutzer nicht die Seite neu laden müssen, wenn eine Datei oder ein Ordner gelöscht wird. Dies wird mit Hilfe der States und Angular-Signals gelöst, welche es ermöglichen Datenänderung im UI direkt sichtbar zu machen.

Design

Für das Design wurde sich ein minimalistisches Layout überlegt. Außerdem war es wichtig eine gewisse Konsistenz für die Wiedererkennbarkeit zu gestalten, dazu werden Farben sowie vorher erwähnte Komponenten wiederverwendet.

Beispiel erste Idee für Landing Page:



Beispiel erste Idee für Erstellungsmenü:

Ressourcen

Eine Liste an Ressourcen, die für die Webanwendung verwendet wurden:

- Ordner-Icon (SVG)
- Bootstrap Icons
- Farbpalette
- Ubuntu Schriftart

BENUTZERDOKUMENTATION

Für die Ausführung der Webanwendung ist eine lokale Entwicklungsumgebung erforderlich, die PHP und MySQL unterstützt. Während der Entwicklung wurde das Paket XAMPP benutzt, dass die benötigten Komponenten Apache, MySQL und PHP zur Verfügung stellt.

Lokale Ausführung

Voraussetzungen

Um das Projekt auszuführen und zu nutzen sind folgende technische Vorrausetzungen erforderlich:

- Serverumgebung mit Apache und MySQL
- PHP ab Version 8.2
- MySQL ab Version 5.7
- Git zum Pullen des Projektes
- NodeJS für Angular
- Moderner Webbrowser

Folgende Versionen wurden während der Entwicklung verwendet:

- PHP 8.2
- MySQL (10.4.32-MariaDB in phpmyadmin)
- NodeJS v23.11.0
- Angular 19.2.11
- Windows 11
- Brave Browser

Installationsschritte

Bevor die Webanwendung gestartet werden kann, muss sichergestellt werden, dass alle oben genannten Vorrausetzungen erfüllt sind. Danach können folgende Schritte ausgeführt werden:

1. Angular CLI im Terminal installieren[1] [2]: npm install -g @angular/cli
2. Projekt-Repository in das Verzeichnis “htdocs/” der XAMPP-Umgebung pullen oder kopieren
   → **Wichtig**: Das Verzeichnis “htdocs/” muss leer sein, es darf keine Dateien oder Ordner beinhalten
3. Im Verzeichnis “File-Manager/Backend/” eine Kopie der Datei “.env.dist” erstellen und diese zu “.env” umbenennen
   → Falls nötig können die Umgebungsvariablen angepasst werden
4. Terminal im Verzeichnis “File-Manager/Frontend/” öffnen
   → NPM-Pakete installieren[3]: npm install
5. XAMPP starten[4]
   → Apache und MySQL aktivieren
6. Im Terminal aus Punkt 4 zurückkehren oder eine neues öffnen (“File-Manager/Frontend/”)
   → Frontend starten[5]: npm start
7. Projekt öffnen
   → Im Browser Frontend öffnen
   → Standardmäßig: http://localhost:4200/
8. Registrierung durchführen

Registrierung und Anmeldung

Die Landingpage präsentiert das Logo mit dem Slogan von Shelfy sowie einen Button, der zur Anmeldungsseite weiterleitet. Dort können sich Benutzer direkt anmelden oder falls sie noch kein Konto haben über den unteren Link “Neues Konto erstellen” zur Registrierungsseite weitergeleitet werden und sich registrieren, indem sie einen Benutzernamen, eine E-Mail-Adresse und ein sicheres Passwort eingeben. Wie vorher bereits erwähnt werden alle Daten sicher verschlüsselt und bei jeder Anmeldung auf Richtigkeit überprüft.

[1] Falls noch nicht installiert
[2] Sollten Fehler auftauchen, als Administrator versuchen. Ansonsten auch folgendes versuchen: https://angular.dev/tools/cli/setup-local#powershell-execution-policy
[3] Falls noch nicht installiert
[4] Sollten Fehler auftauchen, als Administrator neustarten
[5] Falls Port bereits verwendet wird, kann dieser geändert werden: https://stackoverflow.com/a/45358690 in package.json und in der Konfiguration anpassen

Nach erfolgreicher Anmeldung oder Registrierung erfolgt eine Weiterleitung auf die Hauptseite der Webanwendung, nämlich der Speicher.

Fehlermeldungen wie das falsche Passwort werden direkt im Formular angezeigt.

Seiten

Insgesamt gibt es in Shelfy 5 Routen oder Unterseiten. Diese sind:

1. Landing Page über /
2. Login über /login
3. Signup über /signup
4. Storage über /storage
5. Konto über /account

Features

Shelfy bringt viele benutzerfreundliche Features mit sich. Dazu gehören:

- Ordnerverwaltung
    - Erstellen
    - Löschen
    - Öffnen
    - Zum vorherigen Ordner zurückkehren
- Dateiverwaltung
    - Hochladen
    - SHA-256-Signatur kopieren
    - Herunterladen
    - Löschen
- Speicher durchsuchen
    - **Klassisch**: Sucht nur nach Dateien oder Ordner im derzeit ausgewählten Ordner
    - **Erweitert**: Sucht nach allen Dateien und Ordner, die der Benutzer besitzt
- Anzeigeeinstellungen
    - **Datei Signatur anzeigen**: Falls aktiv, wird die SHA-256-Signatur für jede Datei angezeigt
    - **Änderungsdatum anzeigen**: Falls aktiv, wird das Änderungsdatum für Ordner und Dateien angezeigt

- Konto
    - Abmelden

Ist man angemeldet so kann man auf seinen Speicher zugreifen.

Storage ist die Hauptseite der Webanwendung, hier können die obengenannten Features zu Ordnern und Dateien, sowie das Durchsuchen genutzt werden. Im unteren Teil der Webanwendung ist immer eine Bottom-Navigation-Bar zu sehen. Diese gibt die Möglichkeit zu den Kontoeinstellungen weitergeleitet zu werden, ein Menü, zum Erstellen von Ordnern oder Dateien hochladen, zu öffnen, welches nur beim Speicher angezeigt wird, oder wieder doch zum Speicher zurückzukehren.

Auf der Kontoseite kann sich der Benutzer einerseits abmelden aber auch Einstellungen zum Speicher vornehmen. Dazugehören das Suchverhalten, das Ersetz-verhalten bei Dateien, die mit gleichen Namen hochgeladen werden, sowie die Anzeigeeinstellungen zu Dateisignaturen und Änderungsdatum.