Playlist-Builder

Authors: Holzer Tobias, Werlen Nevio

Teacher: Vianin Jeremy

Class: Inf4 App

Inhalt

[1. **Frontend**: 4](#_Toc210224762)

[**1.1** **Ziel:** 4](#_Toc210224763)

[**1.2** **Verwendete Technologien:** 4](#_Toc210224764)

[1.3 **Designprinzipien**: 4](#_Toc210224765)

[**1.4** **Struktur des Frontends** 4](#_Toc210224766)

[**1.4.1** **Aufbau der Seite:** 4](#_Toc210224767)

[**1.4.2** **Hauptbereich** 4](#_Toc210224768)

[**1.4.3** **Footer oder Abschlussbereich** 4](#_Toc210224769)

[**1.5** **Technische Umsetzung** 5](#_Toc210224770)

[**1.5.1** **HTML-Struktur** 5](#_Toc210224771)

[**1.5.2** **CSS-Design** 5](#_Toc210224772)

[**1.5.3** **Interaktive Elemente:** 5](#_Toc210224773)

[**1.6** **Funktionale Aspekte** 5](#_Toc210224774)

[**1.7** **Erweiterungsmöglichkeiten** 6](#_Toc210224775)

[**1.8** **Zusammenfassung:** 6](#_Toc210224776)

[**2.** **Backend:** 7](#_Toc210224777)

[**2.1** **Technologien & Abhängigkeiten** 7](#_Toc210224778)

[**2.2** **Projektstruktur** 7](#_Toc210224779)

[2.3 **Funktionale Beschreibung** 8](#_Toc210224780)

[**2.3.1** **Konfiguration** 8](#_Toc210224781)

[**2.3.2** **Endpunkte (Beispiele)** 8](#_Toc210224782)

[**2.4** **Installation und Ausführung** 8](#_Toc210224783)

[**2.4.1** **Lokale Ausführung** 8](#_Toc210224784)

[**2.4.2** **Docker-Variante** 8](#_Toc210224785)

[**2.5** **Tests** 8](#_Toc210224786)

[**2.6** **Erweiterungsmöglichkeiten** 9](#_Toc210224787)

[**2.7** **Fazit** 9](#_Toc210224788)

[**3.** **Jenkins:** 10](#_Toc210224789)

[**3.1** **Technologien & Setup** 10](#_Toc210224790)

[**3.2** **. Jenkins Pipeline (Jenkinsfile)** 10](#_Toc210224791)

[**3.2.1** **Allgemeine Struktur** 10](#_Toc210224792)

[**3.2.2** **Optionen** 11](#_Toc210224793)

[**3.2.3** **Stages** 11](#_Toc210224794)

[**3.3** **Docker-Konfiguration für Jenkins** 11](#_Toc210224795)

[**3.3.1** **Dockerfile (Jenkins)** 11](#_Toc210224796)

[**3.3.2** **Docker Compose** 11](#_Toc210224797)

[**3.4** **CI/CD Ablauf** 12](#_Toc210224798)

[**3.5** **Vorteile** 12](#_Toc210224799)

1. **Frontend**:
   1. **Ziel:**

Das Frontend ermöglicht das Suchen, Auswählen und Exportieren von Songs in einer Playlist. Es dient als visuelles Interface für ein Musik-Playlist-Management-System.

* 1. **Verwendete Technologien:**
     + HTML5: Strukturierung der Inhalte
     + CSS3 / Flexbox: Styling, Layout und responsive Gestaltung
     + Google Fonts (Inter): Typografie
     + FontAwesome: Icons (z. B. Papierkorb für Löschen)
  2. **Designprinzipien**:
     + Dunkles Theme (Hintergrund #191414)
     + Grüne Akzente (#1db954) für Interaktivität
     + Abgerundete Container und Schatten für Tiefenwirkung
  3. **Struktur des Frontends**

## **Aufbau der Seite:**

* + - Die Seite ist in drei Hauptbereiche gegliedert:
    - Header
    - Enthält den Projektnamen als Überschrift und einen Untertitel.
    - Dient zur Orientierung und visuellen Einführung in die Anwendung.
    - Farblich hervorgehoben durch grüne Akzente.

## **Hauptbereich**

* + - Suchbereich: Eingabefeld für Songs oder Künstler, begleitet von einem Suchbutton.
    - Ergebnisliste / Playlist: Darstellung der ausgewählten Songs mit Titel, Künstlername und Löschoption.
    - Aktionsbuttons: Funktionen zum Hinzufügen von Songs zur Playlist und zum Export der Playlist.

## **Footer oder Abschlussbereich**

* + - In dieser Version nicht explizit vorhanden, potenziell für zukünftige Funktionen wie Navigation oder Zusatzinformationen nutzbar.
  1. **Technische Umsetzung**

## **HTML-Struktur**

* + - Semantische Trennung der Inhalte über Header, Main und Sections.
    - Nutzung von Container-Elementen, um einzelne Funktionsblöcke logisch zusammenzufassen.
    - Eingabefelder und Buttons sind klar voneinander getrennt und mit Klassen für gezielte Styling- und JavaScript-Anbindung versehen.
    - Songs werden als wiederholbare Elemente innerhalb der Playlist dargestellt, was dynamisches Hinzufügen und Entfernen erleichtert.

## **CSS-Design**

* + - Farbschema: Dunkles Theme (Hintergrund dunkelgrau/schwarz) mit grünen Akzenten zur Hervorhebung von Interaktionselementen.
    - Schriftarten: Nutzung der Google-Font „Inter“ für moderne, gut lesbare Typografie.
    - Layout: Flexbox wird für vertikale Ausrichtung und Abstände zwischen Elementen verwendet.
    - Container: Hauptcontainer ist zentriert, mit maximaler Breite von 480 Pixeln, abgerundeten Ecken, Schatteneffekten und leicht transparentem Hintergrund.

## **Interaktive Elemente:**

* + - Buttons und Eingabefelder besitzen visuelle Feedbackmechanismen (Hover- und Fokus-Effekte).
    - Löschen-Buttons für Songs sind rund und mit Icons versehen.
    - Responsivität: Die Anwendung ist flexibel und passt sich an verschiedene Bildschirmgrößen an, insbesondere im mobilen Bereich.
  1. **Funktionale Aspekte**
     + Suchfeld: Ermöglicht die Eingabe von Songtiteln oder Künstlernamen. Die visuelle Rückmeldung bei Fokus unterstützt die Benutzerfreundlichkeit.
     + Playlist: Darstellung der Songs in einer tabellarischen Form mit klarer Trennung zwischen Titel, Künstler und Aktionselementen.
     + Buttons: Einheitliches Design für alle Interaktionspunkte, farblich konsistent und durch Hover-Effekte hervorhebbar.
     + Löschfunktion: Jede Playlist-Zeile besitzt einen dedizierten Button zum Entfernen des Songs aus der Liste.
     + Exportfunktion: Möglichkeit, die zusammengestellte Playlist zu speichern oder herunterzuladen.
  2. **Erweiterungsmöglichkeiten**

Technisch könnte das Frontend noch erweitert werden durch:

* + - JavaScript-Integration: Für dynamische Suchergebnisse, das Hinzufügen und Entfernen von Songs in Echtzeit.
    - Verbesserte Responsivität: Optimierung für Tablets und kleine Smartphones.
    - Animations- und Übergangseffekte: Für flüssige Interaktionen beim Hinzufügen oder Löschen von Songs.
  1. **Zusammenfassung:**

Das Frontend des Playlist-Builders ist modular, semantisch korrekt und responsiv aufgebaut. Es kombiniert moderne Web-Technologien mit einem klaren Designkonzept, das sowohl visuell ansprechend als auch funktional ist. Durch die klare Trennung von Struktur, Layout und Interaktion ist es leicht erweiterbar für dynamische Features oder API-Integration.

1. **Backend:**

Das Projekt implementiert ein Backend-System auf Basis von Flask, das als Schnittstelle zwischen einem Client (z. B. einer Web- oder Mobile-App) und der Spotify Web API dient.  
Hauptaufgabe ist die Authentifizierung über OAuth2 und die Weiterleitung von Anfragen an Spotify, um Benutzerdaten oder Musik-Informationen abrufen zu können.

Das Backend wurde so konzipiert, dass es leichtgewichtig, dockerisierbar und einfach testbar ist.

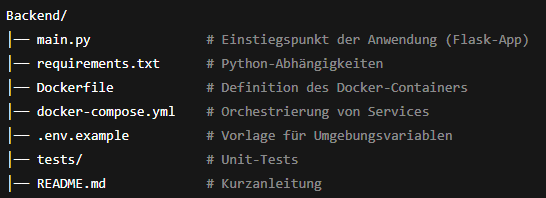
* 1. **Technologien & Abhängigkeiten**

Die wichtigsten eingesetzten Technologien sind:

* Programmiersprache: Python 3
* Framework: Flask 3.0.3 (Bereitstellung der HTTP-Endpunkte)
* HTTP-Requests: Requests – Kommunikation mit der Spotify API
* Konfiguration: python-dotenv – Laden von Umgebungsvariablen aus .env
* Produktionsserver: gunicorn – Bereitstellung des Backends im Container
* Testing: pytest – Unit- und Integrationstests

Alle Abhängigkeiten sind in der Datei requirements.txt dokumentiert.

* 1. **Projektstruktur**



* 1. **Funktionale Beschreibung**

## **Konfiguration**

Über die Datei .env werden zentrale Parameter gesetzt:

* SPOTIFY\_CLIENT\_ID
* SPOTIFY\_CLIENT\_SECRET
* SPOTIFY\_REDIRECT\_URI

Diese Variablen sind notwendig, um den Spotify OAuth2-Flow korrekt auszuführen.

## **Endpunkte (Beispiele)**

Die Flask-App definiert mehrere Routen, unter anderem:

* Login-Start: Weiterleitung des Benutzers zu Spotify, um die OAuth2-Zustimmung zu erteilen.
* Redirect-Handler:
* Empfang des Autorisierungscodes von Spotify
* Austausch des Codes gegen ein Access- und Refresh-Token
* API-Weiterleitungen:
  + Abruf von Benutzerdaten, Playlists oder Musikinformationen über Spotify API
  + Antworten werden im JSON-Format zurückgegeben
  1. **Installation und Ausführung**

## **Lokale Ausführung**

* Repository klonen
* Virtuelle Umgebung erstellen und aktivieren
* Abhängigkeiten installieren:
  + pip install -r requirements.txt
* .env anlegen (auf Basis von .env.example)
* Start der Anwendung:
  + python main.py

## **Docker-Variante**

* Das Projekt ist vollständig dockerisiert:
* .env Datei anlegen
* Container starten:
  + docker-compose up --build
* Das Backend ist standardmäßig unter http://localhost:5000 erreichbar.

# **Tests**

Die Tests liegen im Ordner "backend/tests/" und werden mit "pytest" ausgeführt.

# **Erweiterungsmöglichkeiten**

* Ergänzung zusätzlicher Spotify-API-Endpunkte (z. B. Playlist-Verwaltung, Suche)
* Integration einer Datenbank (z. B. PostgreSQL) zur Speicherung von Tokens oder User-Informationen
* Deployment auf Cloud-Diensten (z. B. AWS, Heroku, Render)

# **Fazit**

Dieses Backend bietet eine skalierbare und containerisierte Lösung, um die Spotify API sicher über OAuth2 einzubinden. Dank Flask bleibt der Code schlank, testbar und leicht erweiterbar.

1. **Jenkins:**

Für das Backend existiert eine CI/CD-Pipeline auf Basis von Jenkins, die in einem Docker-Container läuft.

Zweck dieser Pipeline ist es, das Backend automatisiert zu:

* Bauen
* Testen
* Containerisieren
* Optional in ein Docker-Registry zu pushen

Dadurch wird eine konsistente, wiederholbare und portable Build-Umgebung sichergestellt.

# **Technologien & Setup**

* Jenkins Basis-Image: jenkins/jenkins:lts-jdk17
* Erweiterungen im Dockerfile:
  + docker-ce-cli, git, ssh-client
  + Installation relevanter Jenkins-Plugins (u. a. BlueOcean, Docker Workflow, Git, Pipeline View, etc.)
* Containerisierung via Docker Compose:
  + Start des Jenkins-Masters im Container
  + Zugriff über http://localhost:8080 (Standard-Login: admin/admin)
  + Docker-Socket (/var/run/docker.sock) wird in den Container gemountet → ermöglicht Jenkins, lokale Docker-Builds auszuführen

# **. Jenkins Pipeline (Jenkinsfile)**

## **Allgemeine Struktur**

Die Pipeline ist in stages gegliedert und nutzt das deklarative Jenkins-Pipeline-Format:

* Agent: any (Pipeline läuft auf beliebigen verfügbaren Jenkins-Agents)
* Environment Variablen:
  + IMAGE\_NAME: Docker-Image-Name inkl. Registry
  + REGISTRY: Ziel-Docker-Registry
  + BACKEND\_DIR: Pfad zum Backend (hier backend)
  + DOCKER\_BUILDKIT=1: aktiviert optimierten Docker-Build

## **Optionen**

* ansiColor('xterm'): farbige Logs
* timestamps(): Zeitstempel in den Logs
* disableConcurrentBuilds(): gleichzeitige Builds werden verhindert

## **Stages**

* Checkout
  + Holt den aktuellen Code aus dem Git-Repository (checkout scm).
* Build test image
  + Baut das Backend-Dockerimage mit Ziel test:
  + docker build --target test -t ${IMAGE\_NAME}:test .
* Run tests
  + Startet Unit- und Integrationstests (z. B. via pytest).
* Build final image
  + Erstellt das Produktionsimage (ohne Test-Dependencies).
* Push image (optional, wenn Registry-Zugang vorhanden)
  + Loggt sich in Docker-Registry ein
  + Pusht das fertige Backend-Image (latest + commit-spezifischer Tag).

# **Docker-Konfiguration für Jenkins**

## **Dockerfile (Jenkins)**

Das Jenkins-Dockerfile erweitert das Standard-Image jenkins/jenkins:lts-jdk17 um:

* Docker CLI → Jenkins kann Docker-Befehle direkt ausführen
* Git & SSH-Client → Code aus Repositories holen
* Plugins (z. B. workflow-aggregator, blueocean, git, docker-workflow)

Jenkins wird so vorkonfiguriert, dass kein Setup-Wizard notwendig ist (JAVA\_OPTS="-Djenkins.install.runSetupWizard=false").

## **Docker Compose**

docker-compose.yml definiert den Jenkins-Service:

* Ports 8080 (Webinterface) & 50000 (Agent-Connector)
* Volume jenkins\_home für persistente Daten
* Mount von /var/run/docker.sock → Host-Docker-Integration
* Standard-Admin-Login über Umgebungsvariablen

# **CI/CD Ablauf**

* Der komplette Build-Prozess läuft damit automatisiert ab:
* Entwickler pusht Änderungen ins Git-Repository
* Jenkins startet die Pipeline:
  + Code wird ausgecheckt
  + Docker-Testimage wird gebaut
  + Tests werden ausgeführt
  + Produktionsimage wird erstellt
  + (Optional) Push in Docker-Registry
* Fertiges Image steht bereit für Deployment (z. B. in Kubernetes oder auf einem Server).

# **Vorteile**

* Automatisierung: Keine manuellen Builds oder Tests notwendig
* Reproduzierbarkeit: Gleiche Build-Umgebung durch Docker
* Transparenz: Pipeline ist im Jenkinsfile versioniert
* Erweiterbarkeit: Leicht um Deployment-Stages (z. B. in Kubernetes oder Cloud) erweiterbar