MusicSearch v 2.0 - Dokumentation

Table of contents

1 Projektstruktur: **2**

2 Datenmodell, `models.py` **2**

3 Fachlogik (core) **3**

4 Anbindung externer Systeme (services) **4**
4.1 Die iTunes-Suche 4
4.2 Die MusicBrainz-Suche 7

5 Einstiegspunkt **11**

6 CLI / UI **11**

7 init.py **12**

8 Erweiterbarkeit **12**

Die App ermöglicht dem Benutzer die Eingabe eines Suchbegriffs, der anschließend für die Suche in den Datenbanken von Apple iTunes und MusicBrainz verwendet wird.

Das Projekt ist als Lernprojekt konzipiert und legt den Fokus auf Projektstruktur, Modularisierung und die Trennung von Anwendungslogik und Benutzeroberfläche.

Die Version 2.0 der App erweitert Version 1 und zeigt: - wie man ein Python-Projekt strukturiert, - den Programmcode von der späteren grafischen Benutzeroberfläche trennt, - den Programmcode in Module aufteilt, so dass später Ergänzungen leichter möglich sind - wie die einzelnen Module zusammenwirken und importiert werden, - diese Version enthält bewusst noch keine GUI

1 Projektstruktur:

projectstructure (v2.0.0:)

```
music_search/
├── app/
│   ├── core/
│   ├── services/
│   ├── ui/
│   └── __init__.py
├── main.py
└── requirements.txt
```

Die Verzeichnisse:

- core enthält die zentrale Anwendungslogik und Datenmodelle,
- services enthält die Anbindung an externe APIs,
- ui enthält die Benutzerschnittstelle, hier zunächst nur CLI, später die GUI

2 Datenmodell, `models.py`

Die Datei `models.py`enthält die zentralen Datenmodelle der Anwendung. Diese Modelle beschreiben die Struktur der verwendeten Objekte und sind unabhängig von den benutzten APIs (iTunes, MusicBrainz). Welche Attribute ein Objekt haben soll, welchen Typ die Attribute haben und welche Daten zusammengehören. Hier also beschreibt die Klasse `Track` welche Attribute ein Track-Objekt später hat.

Die Benennung der Datei entpricht gängiger Konvention.

```
1  # models.py
2
3  from dataclasses import dataclass
4
5  @dataclass
6  class Track:
7      title: str
8      artist: str
9      album: str | None
10     source: str  # "itunes"| "musicbrainz"
11
```

Zeile 3: `dataclasses` ist ein Python Standardmodul, das die Erstellung von Klassen erleichtert, die nur Daten und keine Logik enthalten. Ohne `dataclasses` wäre eine Konstruktorfunktion erforderlich:

```
1  class Track:
2      def __init__(self, title, artist):
3          self.title = title
4          ...
```

Zeile 5: `@dataclass` ist ein Dekorator, der kennzeichnet, mit welchem Modul die Klasse erstellt wird.

Zeile 9: `album: str | None` - liefert die Abfrage einen Album-Titel, enthält das Attribut einen String, den Albumnamen. MusicBrainz liefert keine Albumnamen bei der Abfrage, das Attribut enthält dann `NONE`.

3 Fachlogik (core)

Neben dem Datenmodell enthält das Verzeichnis `core` die Orchestrierung der Suchlogik.

```
1  from app.services import itunes, musicbrainz
2  from app.core.models import Track
3
4  def search_all(term: str) -> list[Track]:
5      results: list[Track] = []
6
7      results.extend(itunes.search(term))
8      results.extend(musicbrainz.search(term))
9
10     return results
```

Zeilen 1-2: importieren die Services und das Datenmodell.

Zeile 4: definiert die Funktion `search_all` die als Parameter den Suchbegriff des Benutzers erhält und eine Liste mit Track-Objekten zurückgibt.

Zeile 5: erstellt eine leere Liste und speichet sie in der Variablen `results`. Für eine bessere Lesbarkeit wird eine Typ-Annotation verwendet `:list[Track]`, das ist keine Anweisung, sondern nur eine Information für Mensch und Werkzeuge.

Zeilen 7 und 8: an die zuvor erstellte leere Liste werden mit der Methode `extend()` die Ergebnisse - Rückgabewerte - aus den search-Funktionen der Module `itunes.py` und `musicbrainz.py` nacheinander angefügt.

4 Anbindung externer Systeme (services)

Das Verzeichnis enthält zwei separate Module für die Suche bei iTunes und MusicBrainz.

4.1 Die iTunes-Suche

```python
# itunes.py

import requests
from app.core.models import Track

ITUNES_URL = "https://itunes.apple.com/search"

def search(term: str, limit: int = 5) -> list[Track]:
    params = {
        "term": term,
        "media": "music",
        "limit": limit
    }
    response = requests.get(ITUNES_URL, params=params)
    response.raise_for_status()

    data = response.json()
    tracks = []

    for item in data.get("results", []):
        tracks.append(
            Track(
                title=item.get("trackName"),
                artist=item.get("artistName"),
                album=item.get("collectionName"),
                source="itunes"
            )
        )
    return tracks
```

Zeile 3: importiert das externe Modul `requests`. Es muss gesondert installiert werden z.B. mit

```
pip3 install requests
```

Das Modul stellt eine Schnittstelle für HTTP-Anfragen an Webserver bereit. Aus dem Python-Programm heraus können HTTP-Requests an den Webserver gestellt und dessen Antworten verarbeitet werden.

Zeile 4: importiert die Klasse `Track`.

Zeile 6: definiert eine statische, globale Variable, die die URL der Datenbank aufnimmt.

Zeile 8: definiert die Funktion `search()` die als Parameter den Suchbegriff des Benutzers sowie optional einen int-Wert für die Anzahl der ausgegebenen Ergebnisse. Die Funktion gibt eine Liste mit Track-Objekten zurück.

Zeile 9: erstellt das Dictionary `params`, das die konkreten Suchkriterien enthält. Die keys, die hier eingesetzt werden, entnimmt man der jeweiligen API-Dokumentation.

Zeile 14: die Methode `get()` aus dem Modul `requests` startet eine HTTP-GET Anfrage an den Server, sie erhält als Parameter die URL und die Suchparameter. Die Antwort vom Server wird als Response-Objekt in der Variablen `response` gespeichert.

Das response-Objekt enthält einen Statuscode, einen Header und den Inhalt im json-Format.

Zeile 15: `raise_for_status()` löst eine Exception aus, wenn der Statuscode einen Fehler signalisiert. Dadurch wird bei einem Stuscode mit 4xx oder 5xx, z.B. 404 (Page not found) die Funktion unmittelbar verlassen.

Zeile 17: der Inhalt der Server-Antwort wird mit der Methode `json()` in ein Python-Dictionary (auf oberster Ebene) umgewandelt (geparst) und in der Variablen `data` gespeichert. Diese Dicionary enthält als Wert für den key “results” eine Liste mit Dictionaries:

```
{
    "resultCount": 50,
    "results": [
        {
```

```
            "trackName": "Smells Like Teen Spirit",
            "artistName": "Nirvana",
            ...
        },
        ...
    ]
}
```

Welche Form hier zurückgegeben wird, richtet sich nach dem JSON-Inhalt. Beginnt dieser mit [, so wird von `json()` eine Python-Liste erstellt.

Zeile 20: die for-Schleife iteriert über die Liste "results" des `data`-Dictionary. Also über den Wert der mit der Methode `get()`. `get()` für den key "results" ermittelt wird. Oder als default-Wert (wenn "results" nicht existiert) eine leere Liste `[]`.

ab Zeile 21: füllt die track-List mit Objekten vom Typ `Track`. Die einzelnen Objekte enthalten jeweils die Werte zu den mit `get()` abgefragten keys. So wird z.B. der Wert des keys "trackName" im Attribut `title` gespeichert.

Zeile 29: zurückgegeben wird die Liste `tracks` der Track-Objekte, die dann in die Liste `results` des Moduls `search.py` eingefügt wird:

```
results.extend(itunes.search(term))
```

4.2 Die MusicBrainz-Suche

Die Vorgehensweise weicht bezüglich der Such-Parameter und der verwendeten keys von der iTunes-Suche ab.

```
# musicbrainz.py

import requests
from app.core.models import Track

```

```
MB_URL = "https://musicbrainz.org/ws/2/recording"

HEADER = {
    "User-Agent": "MusicSearchLearningApp/2.0 \
        (info@computer-und-sehen.de)"
}

def search(term: str, limit: int = 5) -> list[Track]:
    params = {
        "query": term,
        "fmt": "json",
        "limit": limit
    }
    response = requests.get(MB_URL, params=params, headers=HEADER)
    response.raise_for_status()

    data = response.json()
    tracks = []

    for item in data.get("recordings", []):
        artist = (item["artist-credit"][0]["name"]
        if item.get("artist-credit")
        else "Unknown")

        tracks.append(
            Track(
                title=item.get("title"),
                artist=artist,
                album=None,
                source="musicbrainz"
            )
        )

```

```
39      return tracks
```

Zeile 8: Anfragen bei MusicBrainz verlangen einen Header mit Informationen über die Herkunft der Anfrage (User-Agent).

Zeile 13-17: hier werden andere keys für die Suchparameter eingesetzt.

Zeile 21: `data`, das Dictionary verwendet andere keys:

```
data = {
    "created": "2024-01-01T12:00:00.000Z",
    "count": 123,
    "offset": 0,
    "recordings": [
        {
            "id": "...",
            "title": "Smells Like Teen Spirit",
            "artist-credit": [
                {"name": "Nirvana"}
            ],
            ...
        },
        ...
    ]
}
```

“recordings” ist der key unter dem man die Ergebnisliste findet.

Zeile 24: in der for-Schleife ist `item` ein Dictionary, der Wert aus der Ergebnisliste:

```
{
    "id": "...",
    "title": "Smells Like Teen Spirit",
    "artist-credit": [
        {"name": "Nirvana"}
```

```
    ]
}
```

Aus diesem werden dann die Werte für das Track-Objekt extrahiert.

Zeile 25: um den `artist` zu erhalten, wird

`artist = item["artist-credit"][0]["name"]`

ausgewertet. Dabei ist `artist-credit` der key des Dictionaries auf oberster Ebene. Der zugehörige Wert zu diesem key ist eine Liste, die wiederum ein key-value-Paar enthält. Mit dem index `[0]` wird auf das erste (und einzige) Element der Liste zugegriffen:

```
"artist-credit": [
        {"name": "Nirvana"}
    ]
```

Weil das Element aus der Liste ein Dictionary ist, mit einem Element :`{"name": "Nirvana"}`, wird der Wert wieder über den key “name” abgerufen.

Eine if-Bedingung stellt anschließend noch sicher, dass “Unknown” ausgegeben wird, wenn es keinen “artist-credit” gibt, der zum Suchbegriff passt:

```
   ...  if item.get("artist-credit") else "Unknown"
```

Besser verständlich ist die if-Bedingung in herkömmlicher Schreibweise:

```
if item.get("artist-credit"):
    artist = item["artist-credit"][0]["name"]
else:
    artist = "Unknown"
```

5 Einstiegspunkt

Die Datei `main.py` ist der Einstiegspunkt für den Programm-Aufruf.

```
1  # main.py
2
3  print("v 2.0.0")
4
5  from app.ui.cli import run
6
7  if __name__ == "__main__":
8      run()
```

Zeile 5: importiert die Funktion `run()` aus dem Modul `cli` im Verzeichnis `ui`. In Zeile 8 wird sie dann aufgerufen.

Zeile 7: `__name__` ist eine eingebaute Python-Variable deren Wert automatisch auf `__main__` gesetzt wird, wenn das Skript direkt gestartet wird. Importiert man die Skript-Datei stattdessen nur (als Modul) in eine andere Skript-Datei, wird `__name__` mit dem tatsächlichen Dateinamen belegt.

6 CLI / UI

Statt einer grafischen Benutzeroberfläche (GUI) verwenden wir zunächst ein Command Line Interface (CLI).

```
1  # app/ui/cli.py
2
3  from app.core.search import search_all
4
5  def run():
6      term = input("Enter search term: ")
7      results = search_all(term)
```

```
8
9       for track in results:
10          print(f"[{track.source}] {track.artist} - {track.title}")
```

Zeile 3: importiert die Funktion `search_all()`.

Zeile 5: definiert die Funktion `run()`, die den Suchbegriff abfragt, ihn an `search_all()` weitergibt und deren Ergebnis in `results` speichert.

Zeile 9: die for-Schleife iteriert über die results-Liste und erzeugt für jeden Eintrag mit `print()` einen Format-String.

7 init.py

`__init__.py` markiert ein Verzeichnis explizit als Paket und erlaubt kontrollierte Imports sowie Initialisierungscode Sie kann leer sein.

8 Erweiterbarkeit

Der modulare Aufbau ermöglicht es, im Verzeichnis `services` den Zugriff auf weitere Datenbanken zu integrieren.

Eine zusätzliche Abfrage für die Anzahl der auszugebenden Ergebnisse bietet sich an.

Weitere Suchparameter könne hinzugefügt werden.

Die Abfrage des Suchbegriffs und die Ergebnisausgabe im Terminal können gegen eine GUI getauscht werden.