SpeakeasySounds API – 3rd Party API Integration and Evaluation

Part 2 – Consuming a 3rd Party API

To extend the functionality and relevance of my API, I integrated the Spotify Web API. This 3rd-party API allowed me to connect my music-sharing backend (SpeakeasySounds) with real-time music data. This was an ideal match for my platform since users are already posting links to tracks and playlists. By connecting with Spotify, I added a layer of music discovery and dynamic content suggestions to enhance user engagement.

API Chosen

I chose the Spotify Web API due to its high relevance to my project. It allows developers to fetch data about tracks, artists, albums, and more. I focused on using two endpoints:

Track Search Endpoint
GET https://api.spotify.com/v1/search?q={query}&type=track
Used to return tracks based on a user's input query.

Client Credentials Flow (OAuth)
POST https://accounts.spotify.com/api/token
Used to generate a temporary bearer token needed to authenticate all requests to the Spotify API.

Authentication Method

Spotify requires all external requests to be authenticated using a token. I used the "Client Credentials Flow," which is designed for server-side applications that don't require access to user accounts.

To retrieve an access token, I had to:

- Register a Spotify Developer App to get a client_id and client_secret
- Encode those credentials in Base64
- Use cURL in PHP to make a POST request to the token endpoint
- Extract the returned access token and use it in the Authorization: Bearer {token} header

How the Endpoints Were Used

I created two files in my /api/external/ folder:

searchSpotifyTrack.php?query=kendrick

This accepts a query string from the user and sends it to the Spotify search endpoint using cURL. It returns an array of tracks including track_name, artist, album, preview_url, and spotify_url.

suggestTracksByTags.php?tags=lofi,chill

This parses the tags from a user's post, combines them into a search query, and fetches relevant Spotify tracks. This suggestion system adds real music recommendations related to posts made by users.

Parameters and Methods

Endpoint	Method	Required Parameters
/searchSpotifyTrack.php	GET	query (e.g. kendrick)
/suggestTracksByTags.php	GET	tags (e.g. lofi,chill)
https://api.spotify.com/v1/search	GET	q, type, optional limit
https://accounts.spotify.com/api/token	POST	Body: grant_type=client_credentials + Headers with Basic Auth

How the Response Was Used

Spotify returns a large JSON response. I used only the most relevant parts:

- name → displayed as track title
- artists[0].name → first artist listed
- album.name → album title
- preview_url → short audio clip
- external_urls.spotify → link to full track

Only the top 5 results are returned for performance and simplicity.

How I Used the API's Documentation

I relied heavily on Spotify's official documentation at:

https://developer.spotify.com/documentation/web-api/

The documentation included:

- Example cURL requests
- Field names in responses
- OAuth token steps
- Required headers and error responses

I referenced their example token exchange to get my authentication working and used the search endpoint example to build my custom PHP wrapper.

Part 3 – Evaluation

Testing with Postman

I used Postman extensively throughout this project to test every endpoint. Postman allowed me to create a Workspace, organize my API into folders, and test both secure and public endpoints.

Workspace & Folder Structure

I created a Workspace called SpeakeasySounds API, with folders for:

- /users – Register, Login
- /posts – CRUD endpoints
- /comments – Comment features
- /likes – Like/unlike endpoints
- /followers – Follow/unfollow
- /external – Spotify integrations

Each folder included all relevant endpoints with working examples. This allowed me to easily switch between endpoints and test various requests.

Request Components Used

For each request, I used the following components in Postman:

- **Method** – GET, POST, PATCH, DELETE
- **URL** – Localhost paths like http://localhost:8888/...
- **Headers** – Especially for secure requests (e.g. X-Access-Token: Bearer {token})
- **Body** – JSON-formatted content for POST, PATCH, and DELETE
- **Params** – For GET requests that needed ?id=, ?tags=, or ?query=

I also tested failure scenarios:

- Sending a request with a missing or wrong token
- Sending invalid JSON
- Searching for a non-existent Spotify term

All error responses were handled with proper HTTP status codes and error messages.

Example Test Flow

To test my API end-to-end, I followed this process:

1. Register a user using /users/registerUser.php
2. Login with /users/loginUser.php and get token
3. Create a post
4. Comment on the post
5. Like the post
6. Follow the post author
7. Search for tracks using /external/searchSpotifyTrack.php
8. Suggest tracks via /external/suggestTracksByTags.php
9. Update and delete post
10. Delete comment
11. Unlike and unfollow

Documentation (MkDocs)

I used **MkDocs** to generate a full developer site:

- Each API module (posts, comments, users, likes, followers) is documented
- Each endpoint includes request methods, headers, sample JSON, and example responses
- The site runs locally with mkdocs serve
- Markdown files were structured and version-controlled via Git

What I Learned

I learned how to:

- Build a modular, secure REST API from scratch
- Authenticate users and restrict access with tokens
- Connect to external APIs using cURL in PHP
- Use documentation to integrate third-party systems
- Organize testing and simulate client requests in Postman
- Create developer-facing documentation using MkDocs

This process showed me how real-world APIs are designed, secured, and documented, and gave me experience that directly applies to professional backend development.