API Systems

Task 1 Research Document

Louisann Borg
BA IDM 6.1

1. Introduction

In today's music industry, streaming platforms and recommendation systems play a large role in discovering new artists and albums. However, metal music fans frequently have difficulty using mainstream music streaming services due to insufficient subgenre-specific filtering, inaccurate metadata, and poor discovery algorithms. Most major platforms, including Spotify, Apple Music, and Deezer, use generic recommendation models that fail to distinguish between the many different metal subgenres.

Metalurgy is an API that aims to transform metal music discovery by providing a genre-specific, AI-based recommendation engine. This API will be useful for music streaming services, independent developers, bands, and metal fans looking for new ways to discover and interact with metal music. This document delves into the research behind RESTful APIs, authorization mechanisms, and security considerations before presenting the Metalurgy API concept in detail.

---

2. RESTful APIs

2.1 What is a RESTful API?

A RESTful API (Representational State Transfer) is an architectural style that facilitates communication between clients (such as web applications) and servers. RESTful APIs are built on standard HTTP methods and stick to stateless principles, making them scalable and easy to integrate across platforms.

2.2 Core RESTful API Principles

- ◊ **Client-Server Architecture***:* Separates front-end applications from the back-end API, allowing for more flexible integration.
- ◊ **Cacheability**: Responses can be cached for improved performance.
- ◊ **Statelessness**: Every API request has to contain all the necessary information, as servers do not store client session data.
- ◊ **Layered Systems**: Multiple layers between the client and the server can exist, these include load balancers or security layers.
- ◊ **Uniform Interface**: This ensures consistent communication using standard HTTP methods and JSON formatting.

2.3 How do API requests work?

When a client application (such as Spotify, a mobile app, or a web service) needs to retrieve or alter data from an API (Application Programming Interface), it uses a structured process to submit a request and receive a response. Here's a step-by-step explanation of how this interaction works:

1. **Client sends and HTTP Request:**
    - The client (e.g., Spotify's app) initiates an HTTP request to the API's endpoint. The request consists of several key components:
        - **HTTP Method** – Specifies the action the client wants to perform (e.g., GET, POST).
        - **URL (Uniform Resource Locator)** – Identifies the API endpoint, which is a specific address where the resource can be accessed (e.g., https://api.music.com/bands?genre=death_metal).
        - **Headers** – Contain metadata such as authentication tokens, API keys, content type, and accepted response format (e.g., Authorization: Bearer XYZ123).
        - **Query Parameters** – Used to filter, sort, or specify data in a request (e.g., ?genre=death_metal to filter bands by genre).
        - **Request Body** – Used in methods like POST, PUT, and PATCH to send data (e.g., JSON data containing new band details).
2. **The API processes the request:**
    - Once the API server receives the request, it goes through the following steps:
        - **Validates the Request –** Checks if the request contains the correct format, authentication credentials, and required parameters.
        - **Retrieves Data from the Database –** If the request is for data retrieval, the API queries the database for the requested information.
        - **Performs Business Logic** – If necessary, the API applies processing rules, calculations, or transformations to the data before returning it.
3. **The API sends a response in JSON format with the requested information.**
    - Once the request has been processed, the API will return a response in a structured format typically JSON (JavaScript Object Notion). The response includes:
        - **HTTP Status Code** – Indicates the success (200 OK) or failure (404 Not Found, 500 Internal Server Error) of the request
        - **Response Headers** – Provide additional information such as content type, caching policies, and API version.
        - **Response Body** – Contains the requested data in JSON format.
4. **The client displays the data (e.g. listing bands, albums etc)**
    - Once the client receives the API response it:
        - **Parses the JSON Data** – Converts the JSON response into a readable format for use in the application.
        - **Displays the Data –** For example, a music app may display a list of bands along with their albums.
        - **Handles Errors** – If the API returns an error response, the client may show an appropriate error message (e.g., “No bands found” or “Invalid request”).

2.4 HTTP Methods

◊ **GET (Retrieve Data)**
- When a client (such as a mobile app, website, or other software) requests data from an API, it uses the GET method. It is the most common HTTP method, and it does not modify any existing data on the server—it simply retrieves it.
- Key Characteristics:
  - **Safe and Read Only:** Since GET requests do not alter the data they are considered safe operations
  - **Idempotent**: Multiple GET requests with the same parameters will always return the same response assuming the data hasn't been changed.
  - **Cached by Browsers:** Web browsers and intermediate servers may cache GET responses to improve performance.
- Real World Example
  - Assume you are using a music streaming app and looking for rock bands. The app makes a GET request to the API, asking for a list of rock bands stored in its database. The API receives the request, retrieves the necessary data, and returns it to your app, which displays the list of bands.

◊ **POST (Create New Data)**
- The POST method is used by a client to send new data to the server for creation. This method is frequently used to add new records to a database, such as setting up a new user account, adding a new song to a playlist, or submitting a contact form.
  - Key Characteristics:
    - **Not Idempotent:** If the same POST request is sent multiple times, it will create multiple new records rather than just one.
    - **Includes a Request Body:** Unlike GET, which retrieved data, POST requires the client to send data to the server in the request body.
    - **Used for Form Submissions and Data Uploads:** Commonly used when submitting forms or uploading files.
  - Real World Example
    - If you create a new account on a social media platform, the app sends a POST request containing your name, email address, and password. The API processes the data, creates a new user in the database, and returns a success message along with the user's unique ID.

◊ **PUT (Update an entire Resource)**

- The PUT method is used by a client to completely replace an existing resource with new data. It updates an existing record by sending a complete copy of the updated resource, even if only one field has changed.
  - Key Characteristics:
    - **Idempotent:** Sending the same PUT request multiple times will always result in the same final state of the resource.
    - **Requires Full Data Replacement:** If a field is missing in the request, it will likely be removed or reset to default values.
    - **Used for Updating Entire Objects:** Typically used when modifying entire records in a database.
  - Real World Example
    - Take for example a music band management app where a user wants to update a band's profile. If the user changes the genre from "Rock" to "Alternative Rock," the app sends a PUT request with all the band's details, including its name, albums, and new genre. The API updates the record, replacing the old information with the new data.

◊ **PATCH (Partially update a Resource)**

- The PATCH method is similar to PUT, but it only updates specific fields rather than the entire resource. It is more efficient to make small changes to an existing record.
  - Key Characteristics:
    - **Not Idempotent:** While it often behaves idempotently, multiple PATCH requests may have different outcomes depending on the updates being applied.
    - **Used for Minor Updates:** Unlike PUT, which requires sending a complete dataset, PATCH allows updating just one or a few fields.
    - **Efficient and Reduces Data Transfer:** Since it only updates the changed fields, it requires less data transmission.
  - Real World Example
    - Imagine an e-commerce website where a customer wants to update only their phone number without modifying their full profile. Instead of sending the entire profile data (as with PUT), the app sends a PATCH request containing just the new phone number. The API updates only that specific field in the database while keeping the rest of the information unchanged.

◊ **DELETE (Remove Data)**

- The DELETE method is used when a client requests that a specific resource be removed from the database or server.
  - Key Characteristics:

- **Idempotent:** Sending the same DELETE request multiple times will always result in the resource being deleted (or already being deleted)
- **Irreversible in Most Cases:** Once a resource is deleted, it typically cannot be recovered unless the system has a backup or undo mechanism.
- **Often Requires Authentication:** APIs may require users to be logged in and have appropriate permissions to delete data.

▪ Real World Example

- When a user decides to delete a post from a social media platform, the app sends a DELETE request to the API. The API verifies permissions, deletes the post from the database, and returns a confirmation message. The post will no longer be displayed on the user's profile or feed.

---

3. Authorization & OAuth 2.0

3.1 *What is OAuth 2.0*

OAuth 2.0 is an industry-standard authorization framework that allows secure API access without exposing user credentials like passwords. It is commonly used by applications that must interact with third-party services on behalf of users. Instead of requiring users to share their login information, OAuth 2.0 provides limited access via access tokens issued following user consent.

OAuth 2.0 enables a user to authorize one application (the client) to access data hosted by another service (the resource server) without disclosing their actual credentials. This is accomplished through a series of redirects and permission requests, ensuring security while allowing for seamless integrations between applications.

3.2 *Benefits of OAuth 2.0*

OAuth 2.0 offers several advantages in the realm of authentication and authorization:

◊ **Secure Authentication:** Ensures that API requests are authorized only by authenticated users, reducing risks of credential theft and unauthorized access.
◊ **Token-Based Access:** Users receive an access token instead of sharing their credentials. These tokens have expiration times, ensuring that access is only temporary and reducing security risks.
◊ **Granular Permissions:** OAuth 2.0 allows defining specific permissions (scopes), so users can limit the type of data that third-party applications can access.
◊ **Third-Party Integration**: Enables seamless connectivity between different platforms and services, such as Google, Spotify, GitHub, and more, without compromising user security.

◊ **Single Sign-On (SSO) Support**: Users can authenticate once and grant access to multiple applications without having to log in multiple times.

3.3 *OAuth 2.0 Components*

OAuth 2.0 consists of several key components that facilitate secure authorization:

◊ **Scopes**: Define the level of access granted to the application. For example, a music streaming API might use scopes such as read_bands (to read user's favorite bands) or modify_playlists (to allow modifying a user's playlist).
◊ **Access Tokens**: These are short-lived credentials issued by the authorization server that allow clients to access the protected resources. Once expired, a refresh token (if provided) can be used to obtain a new access token.
◊ **Client ID & Client Secret**: When a third-party application wants to use OAuth 2.0, it must register with the authorization server, which then issues a unique Client ID (public identifier) and a Client Secret (private key) for authentication.
◊ **Authorization Server:** This is responsible for authenticating the user, obtaining consent, and issuing access tokens. It ensures that only authorized applications can access user data.
◊ **Resource Server:** Hosts the protected resources and validates access tokens before providing requested data to the client application.
◊ **Redirect URI:** The URL where users are redirected after granting or denying access. It is pre-registered to ensure security and prevent redirection attacks.

Using OAuth 2.0, developers can create secure authentication and authorization workflows that protect user privacy while ensuring seamless integration between services.

---

4. API Security Considerations

4.1 *Overview of OWASP API Security Top 10*

The OWASP API Security Top Ten is a comprehensive list of the most serious security threats that modern APIs face. These flaws can result in serious consequences such as data breaches, unauthorised access, and service disruptions. APIs continue to play an important role in connecting applications and facilitating data exchange, so developers, security professionals, and organisations must be aware of the risks. To protect sensitive information and prevent cyber threats, API designers must follow industry best practices such as strong authentication and authorisation mechanisms, validating user inputs, and securing data in transit and at rest. Businesses can improve API resilience and protect themselves from potential attacks by proactively addressing these security concerns.

4.2 *Two Major Security Threats & Mitigation Strategies*

1. **Broken Object Level Authorization (BOLA)**

   ◊ **Threat**: Attackers could manipulate API requests to gain access to or modify other users' data.
   ◊ **Solution**: Implement role-based access control (RBAC) to limit access to specific actions.

2. **Security Misconfiguration**

   ◊ **Threat**: API default settings or misconfigured endpoints may expose confidential data.
   ◊ **Solution**
      - Enforce **HTTPS** encryption.
      - Store API keys as **environment variables** instead of hardcoding them.
      - Disable debug endpoints in production environments.

---

5. Proposed API Concept: Metalurgy

5.1 *Overview*

Metalurgy is an API that enhances the discovery and classification of metal music. Unlike other APIs, Metalurgy offers deep metadata tagging, AI-powered recommendations, and community-driven insights.

5.2 *Key Features*

◊ **Band & Album Search**: Helps discover new artists by genre, country, or popularity
◊ **Subgenre Classification**: Categorizes bands based on niche metal styles (e.g. Blackened Death Metal, Technical Trash)
◊ **AI-Powered Recommendations**: Suggest artists based on user behaviour.
◊ **Community Driven Tagging**: The user contributes metadata and genre labels.

5.3 *Unique Selling Points (USPs)*

◊ **Genre-Specific Accuracy**: Unlike Spotify, Metalurgy focuses solely on metal subgenres.
◊ **Developer-Friendly**: RESTful API with clear documentation.
◊ **AI-Powered Personalization**: Uses machine learning to refine recommendations.
◊ **Open-Source Contribution**: Community-driven improvements to API datasets.

5.4 *Target Audience*

◊ **Metal Music Fans**: Looking for better discovery options.
◊ **Indie Bands and Labels**: Seeking exposure for underground artists.
◊ **Developers**: Creating bots, websites, or streaming applications.

5.5 *Example API Requests*

1. **Retrieve Bands by Sub-Genres**

   ◊ GET / bands?genre=doom_metal

2. **Fetch Recent Album Releases**

   ◊ GET /albums?year=2024

3. **Post a new album review**

   ◊ POST /reviews { “albums”: “Ghost Reveries” , “rating”: 9}

5.6 *Implementation Plan*

1. **Phase 1: Research and Planning**

◊ Conduct market analysis on metal music discovery gaps.
◊ Define API endpoints, authentication mechanisms, and security measures.

2. **Phase 2: Development**

◊ Build a prototype of the Metalurgy API.
◊ Implement OAuth 2.0 authentication and API security measures.
◊ Integrate AI-powered recommendation features.

3. **Phase 3: Testing & Refinement**

◊ Conduct beta testing with early adopters (metal music fans, developers, and bands).
◊ Gather feedback on usability, speed, and accuracy of recommendations.
◊ Optimize API response times and improve metadata categorization.

5.7 *API Workflow*

1. **Client App (User Interaction)** – The user interacts with Spotify, Bandcamp, or a Discord bot.
2. **API Request** – The app sends a request, e.g., GET /recommendations?genre=doom.
3. **Metalurgy API (Processing)** – The API filters and processes the request.
4. **Database & AI Engine** – It retrieves metadata, applies AI-driven filtering, and refines recommendations.
5. **API Response** – The API sends a JSON response (e.g., {band: ‘Gojira})

---

6. Conclusion

Metalurgy will bridge the gap in metal music discovery by providing a RESTful API with secure OAuth 2.0 authentication and strong security measures. This API, which uses AI-

powered recommendations and community-driven metadata, will empower developers, bands, and fans alike. The following steps include development and testing to ensure a consistent user experience.

References

- ◊ Fielding, R.T. (2000) *Architectural styles and the design of network-based software architectures*. University of California, Irvine.
- ◊ Hardt, D. (2012) *The OAuth 2.0 Authorization Framework*. Internet Engineering Task Force (IETF). Available at: https://tools.ietf.org/html/rfc6749 (Accessed: 10 March 2024).
- ◊ OWASP (2023) *OWASP API Security Top 10*. Open Web Application Security Project. Available at: https://owasp.org/www-project-api-security/ (Accessed: 10 March 2024).
- ◊ Richardson, L. and Ruby, S. (2007) *RESTful Web Services*. O'Reilly Media.
- ◊ Okta (2023). OAuth 2.0 and OpenID Connect. Available at: https://developer.okta.com/docs/concepts/oauth-openid/ (Accessed: 25 March 2025).
- ◊ Auth0 (2023). What is OAuth 2.0?. Available at: https://auth0.com/docs/authenticate/protocols/oauth (Accessed: 25 March 2025).
- ◊ Google Developers (2024). OAuth 2.0 for Authorization. Available at: https://developers.google.com/identity/protocols/oauth2 (Accessed: 25 March 2025).