API Research Document

Part 1 – RESTful APIs (KU1)

A RESTful API is an API that follows the rules of REST, which stands for Representational State Transfer. It is a way to connect different systems over the internet using standard HTTP methods like GET and POST. REST APIs are used because they are simple, fast, and can work with many types of software.

There are some core principles that make an API RESTful:

- Client-Server Architecture: The client (like a mobile app or website) and the server (which stores and manages the data) are separate. They communicate using requests and responses.
- Statelessness: Each request from the client must have all the information the server needs. The server doesn't remember previous requests.
- Cacheability: Responses can be stored (cached) to improve performance.
- Layered System: The API can have multiple layers (like a security layer) between client and server, but they don't affect the way requests are made.

When making a request to a REST API, the client sends a URL with a method like GET, POST, PUT, PATCH, or DELETE, based on what it wants to do. For example:

- GET gets data from the server
- POST sends new data to the server
- PUT/PATCH updates existing data
- DELETE removes data

The request might also have a Query String (like ?user=alex) or a Request Body (used in POST/PUT) with JSON data.

In my case, I want to create an API where users can upload songs, follow each other, post music-related content, and react to tracks. For example:

- GET /tracks – gets a list of uploaded tracks
- POST /tracks – uploads a new track (e.g. { "title": "My Song", "artist": "Alex", "file": "song.mp3" })
- GET /users/:id/followers – shows a user's followers
- POST /reactions – reacts to a song (e.g. { "track_id": 12, "emoji": " 🔥 " })
- POST /posts – create a new post about a song (e.g. { "text": "check this out lmao", "track_id": 8 })
- POST /comments – comment on a track or post (e.g. { "post_id": 5, "text": "hell yeah" })

This API can be used by external apps, like a mobile music player or a small community website.

Part 2 – Authorisation (KU2)

For this API, I would likely use OAuth 2.0 for authorisation. This system is used by big platforms like Spotify and Google because it is secure and flexible, and it is the standard for authentication.

OAuth 2.0 lets users log in or connect with a service without giving their password to every app. Instead, users log in with their main account, and the app gets an Access Token. This token lets the app use the API in a limited way, depending on what the user allowed.

Some important terms in OAuth 2.0:

- Scopes: These define what the app is allowed to do.
  For example, read:tracks lets the app view tracks, but not upload new ones. Other examples could be write:posts for creating posts or follow:users for following someone.
- Access Token: A string that the client sends with each request to prove it is allowed to use the API.
  The token usually looks like a long random string like ya29.a0AfH6SMA5...
- Client ID and Client Secret: These are given to every third-party app. They identify the app when it talks to the API.
  For example, an app called "BeatShare" might have:
  - Client ID: beatshare123
  - Client Secret: s3cr3tK3yXyZ
    These are used when the app first requests an access token and must be kept safe, especially the secret.

In my system, I will implement OAuth so that people can use their existing accounts (like Google or Spotify) to log in and use the music API without creating a new account. When a user logs in, they will get an access token. All protected routes, like uploading a song or reacting to a post, will check this token.

Part 3 – Security Considerations (KU3)

I looked at the OWASP API Security Top 10, which lists the biggest risks for APIs. These are updated regularly by people. Some of the current top threats include:

- Broken Object Level Authorization (BOLA) – This happens when users can access other people's data by guessing IDs (like /tracks/123).
- Broken Authentication – This is when the login system is weak or tokens are not properly validated.
- Excessive Data Exposure – The API sends more data than needed, which might include private information.
- Rate Limiting Issues – If there's no limit, attackers can spam the API with thousands of requests.

For my API, I will focus on fixing these two:

1. Broken Object Level Authorization (BOLA)
I would make sure that users can only access their own resources or public ones.
For example, when getting /users/123/tracks, the system will check if the requester is user 123 or has permission.
If not, it will return a 403 Forbidden error.

2. Rate Limiting
I would add rate limiting so each user or IP address can only send a limited number of requests per minute (e.g. 60).
This will stop bots from overloading the system or trying to guess tokens.

I would also use HTTPS to encrypt all data sent between client and server, and I will validate all input to avoid injections.

This would be an ideal way to set up an app that combines social media and music sharing. It uses RESTful API structure to keep everything organized and simple, includes OAuth 2.0 for secure logins, and takes care of important security issues like authorization and rate limiting. It's a solid base for any application or system where users need to post content, follow others, and interact with music in different ways.