Using a RESTful API to get data from a music database

Catching up on REST

Craft the perfect URL to access metadata in an online music database. *By John Cofield*

Many network-based web services offer APIs that comply with the REpresentational State Transfer (REST) design style. APIs that comply with REST constraints are called RESTful.

RESTful APIs are attractive because they leverage existing HTTP infrastructure, the same infrastructure and protocols that browsers use. As a result, a RESTful API has access to a great variety of resources (applications, libraries, developers, etc.). From an end-user perspective, you can use your existing browser as a client to access server databases. Developers like RESTful APIs because it is easy to integrate server data and resources into applications.

Music services often use RESTful APIs to provide metadata about artists and musical tracks. Offering this data through a REST interface makes it very easy to build a client application that will receive and display the information. This article shows how to query music data through a RESTful API.

Why REST

A RESTful API allows the client to retrieve data from the server without needing to know details about the server back-end implementation. There is no need for the client to know which server or database management software is used. In addition, the API enhances security by restricting who can access data and selectively restricting which data is accessible.

RESTful APIs operate in two modes, request and response. The client sends a

More on REST

RESTful APIs are built around the following design principles:

- User interface – all requests for the same information look the same, regardless of where the request came from.
- Client-server decoupling – client and server as assumed to be totally independent. The only information that passes between them is through the API.
- Statelessness – each request contains all the input necessary for processing. There is no need to establish a "session." The state of the server remains constant.
- Cacheability – data should be cacheable on the client or server to improve performance.
- Layered system architecture – intermediate layers are integrated invisibly into the architecture. The client can't tell if it is connected directly to the end server or through an intermediary. Layered components adhere to REST principles.
- Code on demand (optional) – integration with client-side components such as Java-applets or JavaScript.

An API isolates the client from the server, allowing communication through a uniform interface. This approach facilitates the development and debug process by giving you better visibility into network components. As a result, implementation of changes on either the client side or the server side can occur without either one affecting the other, as long as the interface protocol is followed [1].

The following is a list of the REST interface constraints:

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of the application state

The first REST constraint, "identification of resources," uses identifiers to identify target resources. In a RESTful API, these identifiers are referred to as representations. By design, a RESTful API never manipulates server resources directly. Instead, it only manipulates their representations, the identifiers. In the example in this article, you will see how these representations are used in communication between client and server.

Photo by Aleksandar Cvetanovic on unsplash

request to the server for data. The server then responds with a status code, and if appropriate, the requested data.

In a RESTful web service, requests made to a resource's URI elicit a response with a payload usually formatted in HTML, XML, or JSON. The most common data transfer protocol for these requests and responses is HTTP, which provides operations (HTTP methods) such as `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`. See the box entitled "More on REST" for additional background on RESTful APIs.

Music APIs

Several well-known music services provide RESTful API database access to musicians and to partners who want to integrate their content or music into commercial products.

HTTP Primer

In order to understand RESTful APIs, it is essential to understand the role that HTTP plays. The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems [3]. HTTP was designed, from its beginning in 1990, to support the client-server model (Figure 1), with requests passing in the form of messages from a client to a server. A response message and status code then passes from the server to the client.

When HTTP was first introduced, the only method used to send requests was the `GET` method. HTTP has since evolved to include additional methods to allow requests to modify and delete content on the server. Among the methods currently used are: `GET`, `POST`, `PUT`, and `DELETE` (Table 1).

When a request message is sent by a user agent (the client application) to a server, the message typically contains header fields that can include metadata such as: method, target hostname, content type, content length, and other characteristics of the client, host, or message.

Upon receiving a request from a user agent, the server determines whether it can accept the request. If accepted, the server responds by sending a status code and a message back to the client. Response message content might include metadata or the requested target resource.

Spotify, for example, provides an API that allows hardware partners to develop applications for home audio systems, music players, headphones, and other Internet-enabled devices. According to the Spotify for Developer website, "Spotify Web API endpoints return JSON metadata about music artists, albums, and tracks, directly from the Spotify Data Catalog."

SoundCloud is a music service that allows musicians to share their music with a community of artists and listeners. Musicians can use the API to upload and manage their music for listeners.

MusicBrainz views its service as an open encyclopedia for music metadata. MusicBrainz is modeled after Wikipedia in that it is community-driven [2]. The metadata content is primarily, but not exclusively, targeted at music player and tagger applications. In this article, I will use the open source MusicBrainz API to request and receive music data through HTTP. (See the "HTTP Primer" box for more on accessing resources through HTTP.)

MusicBrainz API

The MusicBrainz API gives the client access to a wide range of music metadata about artists and their music, including biographical information, release dates, and media formats. Requests are in the form of a URL that is comprised of multiple components, some mandatory and some optional [4].

REST constraints require the API to be stateless. This means that each request sent by the client to the server must contain all the information the server needs to respond appropriately. As a consequence, all the necessary information is contained in the URL. In the case of MusicBrainz, the syntax for the URL is as follows:

```
<api_root><entity><mbid><inc><format>
```

`<api_root>` is the partial URL to the API, in this case:

```
https://musicbrainz.org/ws/2/
```

`<entity>` is one or more information categories, such as artist, recording, release, etc.

`<mbid>` is the MusicBrainz identifier that is unique to each target resource in the database.

`<inc>` is an optional subquery string.

`<format>` is an optional format string that specifies the transfer format, which can be either JSON or XML. (XML is the default format.)

Python urllib Module

This example illustrates one approach for sending an HTTP or HTTPS request and retrieving the response. I will use the `GET` method to retrieve metadata for a single recording and will request the response data in JSON format.

Multiple Python packages are available to send HTTP requests and handle responses. Some have external package dependencies. For the purposes of this exercise, I will use the built-in *urllib* module [5], which does not require installing an external package.

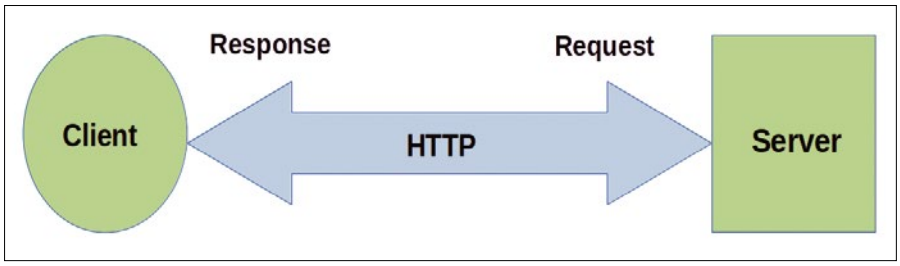

Figure 1: HTTP supports the client-server model.

Table 1: HTTP Methods

`GET`	Asks the server to send the target resource based on an identifier (representation).
`POST`	Asks the server to have the target resource process the enclosed content specified by an identifier.
`PUT`	Asks the server to have the target resource create or replace the enclosed content specified by an identifier.
`DELETE`	Asks the server to have the target resource specified by an identifier to be deleted.

John: Should "equest" in the code below be "request"? -- ap

The goal of the exercise is to:

- Send a request from the client (Python script)
- Retrieve the response data from the MusicBrainz server
- Retrieve the response status code

I'll start by importing the required Python modules:

```
import json
import os
import urllib.request
import webbrowser
import pprint
```

I'll use the following Python methods and objects:

```
urllib.request.Request()
json.loads()
equest.get_method()
webbrowser.open_new_tab()
request.host
request.status
request.type
```

URL Components

Listing 1 shows the values assigned to the MusicBrainz URL components. Each component is concatenated to form the full URL (`api_url`) required to request the API resource.

In Figure 2, I request a resource using the `urllib.request.Request(api_url)` method. Since the only argument is `api_url`, the request method defaults to the `GET` method. A `POST` request is sent to the server using the same method, specifying `POST` or passing a `data` argument. However, the `POST` method must be explicitly specified.

In Figure 3, you can see the response values displayed in the Python IDLE Shell window. Responses normally contain a lot of information, but I have limited the output to display only the `type`, `host`, and `method` to confirm that the request was sent to the MusicBrainz server, that it was sent via secure HTTPS, and that the `GET` method was used.

Note that the JSON data is deserialized by the `json.loads()` method, converted to Python dictionary form, and displayed. Finally, the code value `200` indicates that the request was successfully executed.

If you prefer viewing the data in a browser, one option you can use is the `open_new_tab()` method from the Python built-in module `webbrowser`. The web browser output is shown in Figure 4.

Security

Web APIs typically require some sort of authentication to verify that the client submitting a request is authorized to do so. The most common authentication protocols are HTTP Digest Access Authentication and JSON Web Token (JWT). HTTP Digest Access Authentication uses 256- or 512-bit hash-encrypted username and password to authorize access. JWT tokens are often generated by a third-party authorization server that

Listing 1: URL Component Values

```
api_root = "https://musicbrainz.org/ws/2"
entity = "/recording"
mbid = "/b97670e0-08fe-42fe-af39-7367a710c299"
jfmt = "?&fmt=json"

# Full API URL
api_url = api_root+entity+mbid+jfmt
```

```
import json
import os
import urllib.request
import webbrowser
import pprint

api_root = "https://musicbrainz.org/ws/2"
entity = "/recording"
mbid = "/b97670e0-08fe-42fe-af39-7367a710c299"
jfmt = "?&fmt=json"

# Justice's Groove URL

api_url = api_root+entity+mbid+jfmt

def mbz_recording(api_url):

    request = urllib.request.Request(api_url)
    print('request.type: ', request.type)
    print('request.host: ', request.host)
    print('request.get_method(): ', request.get_method())
    with urllib.request.urlopen(request) as response:
        data = json.loads(response.read().decode("utf-8"))
        statcode = response.status

    # Pretty Print
    pp = pprint.PrettyPrinter(indent=4)
    pp.pprint(data)
    #
    print('Print response status:\n', statcode)
    webbrowser.open_new_tab(api_url)

    return

mbz_recording(api_url)
```

Figure 2: Requesting data with a Python script.

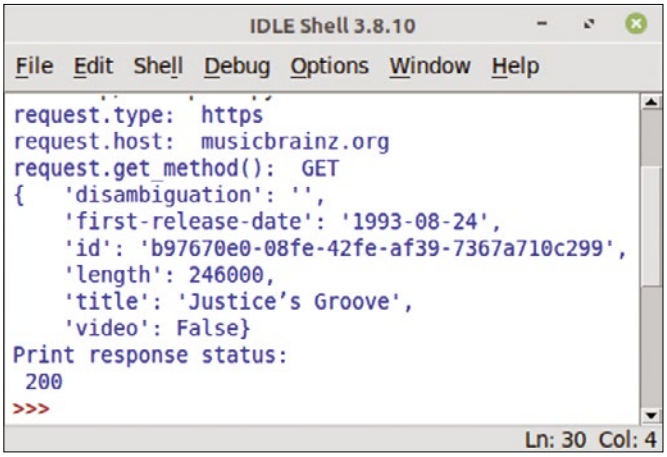

Figure 3: Server response: JSON data text output in the Python IDLE Shell window.

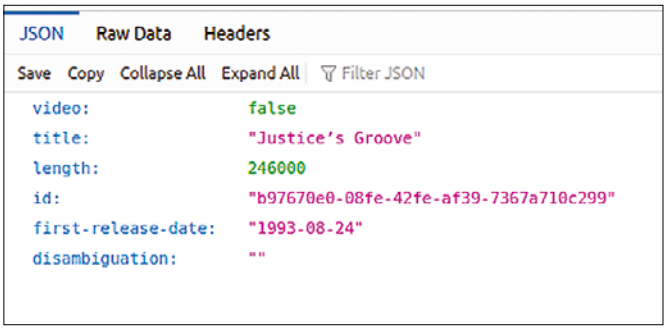

Figure 4: Server response: JSON data to the browser.

you authorize in advance to give you access to one or more applications. The token approach gives you a single sign-on (SSO) source so you don't have to give your username and password to the web service.

MusicBrainz `POST` requests, which allow you to alter data, require authentication. In order to submit a `POST` request, a client application must register using either a username/password or JWT (OAuth 2.0) token. An authorization request must be submitted prior to submitting a `POST` request. See the MusicBrainz website for more information on MusicBrainz authentication [6].

Summary

A RESTful API is an easy way to access music data for a custom script or client application. The REST architecture enables you to develop client applications that can reach flexible and secure RESTful APIs. REST lets you update client and server components independently, which makes RESTful components easier to maintain than tightly integrated client/server systems, and the ability to manipulate representations rather than resources adds additional security to the design. ■■■

Author

John Cofield is a retired software marketing manager in Northern California. His training is in electrical engineering, and he has worked at multiple Silicon Valley semiconductor and software companies. His nontechnical interests include Jazz music ranging from Modal to Fusion.

Info

[1] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*, PhD Dissertation, University of California, Irvine, September 2000: *https://roy.gbiv.com/pubs/dissertation/top.htm*

[2] MusicBrainz: *https://musicbrainz.org/*

[3] HTTP Semantics, RFC 9110: *https://httpwg.org/specs/rfc9110.html*

[4] MusicBrainz FAQ: *https://musicbrainz.org/doc/MusicBrainz_API#General_FAQ*

[5] urllib: *https://docs.python.org/3.10/library/urllib.html*

[6] MusicBrainz authentication: *https://musicbrainz.org/doc/Development/OAuth2*