REST API Crash Course

Introduction to REST APIs

These are the notes to support the REST API Crash Course. We're going to first focus on the concepts of REST APIs and then apply these concepts by building our own.

An API (application programming interface) is a setup to allow two pieces of software to communicate.

One of the challenges with software communicating with each other is that every software system is a little different, so it helps if they're communicating using some standard notation. This notation is usually JSON.

JSON is a standard notation (think of it as a language, in a way) used to communicate between software. XML is also a popular notation to use, but is a lot less favorable compared to JSON and I believe it is fading out in new software projects.

"JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence."

https://www.json.org/json-en.html

This communication between software is typically one direction. This means one software system will do all of the talking, and the other software system will request data. You could say **server** (backend) and **client** (frontend).

A very common setup for this is a data processing server and a website client. The client is going to make **requests** to the server.

Let's say we are building an app to store information about drinks, and we want to get information about a particular drink. I'll likely have a server, let's say written in Python, that manages this data by connecting to a database. The client can then request this data from the server, the server gets the information from the database, and then returns the data to the client.

This software doesn't just open up everything for other software to use. Instead, the developers carefully choose specific things that should be exposed for other software to consume. These things are exposed as API endpoints.

So for the backend we might create an API endpoint and it will look like this: **/drinks/<id>** where the id could be any ID to grab information about a particular drink.

Now you might be asking, What's with the slashes? That's where the REST part of this comes in. **REST** (representational state transfer) is a particular type of API that allows the transfer of data (also known as **state**, as the acronym would imply) over the internet. So these slashes are actually a URL.

The full endpoint might be used like so: calebcurry.com/drinks/5.

Why the Complexity?

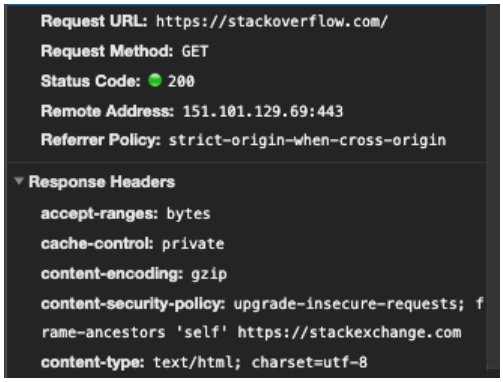
You may ask yourself, instead of having a front end talk to a back end which talks to the database, why don't you just simplify and remove the middle man (the server)? Here are a few reasons:

- **Security** - The most popular front end language is JavaScript, which in front end web development has no concept of security. On a side note, we can include authentication and authorization with our API so only certain people using the front end applications can access sensitive data.
- **Versatility** - with a single backend, we can build numerous front ends that have the single purpose of presenting and interacting with data. This means we could have a website, a mobile application, and a desktop application, and they're all going to work properly with the same data because the data processing is all done on the back end.
- **Modularity** - We can swap out the backend without having anyone notice or have to update an application. As long as the backend exposes the same API. This is a perfect example of **abstraction**. We are abstracting away the data processing and always working with a consistent interface (a REST API that uses JSON).
- **Interoperability** - If desired, your API can be public for any developers to consume. This means that the frontend and backend do not have to be by the same developer. There are tons of public APIs out there that we can use to create cool apps. You could make an instagram browser, or a cryptocurrency trading bot, or a machine learning model based off of YouTube analytics.

REST API Methods

When you open Chrome or Firefox or your browser of choice, and you go to a website like stackoverflow.com,behind the scenes you're making a **GET** request. And the website is listening for requests at the root (/) and returns HTML.

You can see all of this in the developer tools of any modern browser.



Notice the request method is GET, and the content-type is text/html.

To get JSON, there is actually another web address that you can use, api.stackexchange.com, that has endpoints that return JSON.

GET is the first method you should know about.

POST is another method, which is used when you submit forms on web pages. APIs also support POST and this is the method you usually use when you want to add or modify data, such as adding a new answer to a question on Stack Overflow.

There are a few more you should know about, which we will get into later.

Consuming an API

We are going to work with the Stackoverflow API available at api.stackoverflow.com.

We can make a request by following through to an example and copying the URL.

Here's an example:

https://api.stackexchange.com//2.2/answers?order=desc&sort=activity&site=stackoverflow

Order ,sort, and site are examples of **query string parameters**. These are sent to the server and are used to customize the response from the server.

This is a GET request as we can see in the headers of the request in developer tools, but now the response in JSON.

There are also POST options in this API as well, but these require authentication.

For more elaborate API consumption, I recommend Postman, which is a fairly standard tool in any web developers toolbox. This allows you to make similar requests to what we just did, but is fully customizable with the headers and body.

Later on when we start sending data to an API, this will come in handy, because we can build out objects in the body of the request. This allows us to send data in a similar way to how we had parameters in the URL, but now we can work with more sophisticated data by using JSON.

Consuming an API with Python

```
pip3 install requests
```

Thankfully, JSON is directly convertible to a Python dictionary (group of key-value pairs).

```
import requests
import json

response =
requests.get("https://api.stackexchange.com/2.2/questions?order=desc&sort=a
ctivity&site=stackoverflow")

for data in response.json()['items']:
    print(data['title'])
    print(data['link'])
    print()
```

This would be a good time to research what standard JSON looks like.

Creating a basic API

Flask is nice because it has a really simple setup that you can build a web page with just a few lines of code. I show this in action in another video on deploying Flask apps to AWS.

We are going to start building our application as a module (a single file), but for more sophisticated applications it's nice to follow a pattern. Specifically the one from here. It describes the setup to build your application as a package instead of a module.

In Python, a **module** is one file while a **package** is many (a collection of modules).

Starting in your folder that you want all your files, execute these in the terminal:

```
python3 -m venv .venv
source .venv/bin/activate
pip3 install flask
pip3 install flask-sqlalchemy
pip3 freeze > requirements.txt
touch application.py
```

This will create a virtual environment and activate it, install the dependencies we need, list them in a file requirements.txt, and then create a new file for our code called application.py. We can write code now, but keep your virtual environment running.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello!"
```

Now back to the terminal (still in the virtual environment)

```
export FLASK_APP=application.py
export FLASK_ENV=development
flask run
```

You'll need to do these exports every time you start your terminal window. So always make sure to

1. Be in the right directory
2. Activate your virtual environment
3. Run your exports

```
class Drink(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.String(120))

    def __repr__(self):
        return f"{self.name} - {self.description}"
```

```
python3
from application import db
db.create_all()
from application import Drink
drink = Drink(name="Grape Soda", description="Tastes like the real thing")
db.session.add(drink)
db.session.add(Drink(name="Cherry", description="Tastes like that one ice
cream"))
db.session.commit()
Drink.query.all()


exit()
flask run
```

OTHER API Methods

DELETE is used to delete a resource. Simple enough.

PUT replaces a resource. It is similar to POST but with a minor (but important) difference. PUT requests are said to be idempotent. This is a fancy word to say that if you do the exact same request with PUT numerous times, it has the same effect. This is different from POST as POSTing the same thing numerous times is not guaranteed to not result in duplicates. That's why if you refresh a page after you fill out a web form, it may warn you that you'll be re-submitting the form and can cause side effects.

This description is based on the definition of POST and PUT, but as a developer building an API, you're still required to implement the idempotent behavior of PUT.

I like this answer on Stackoverflow.

If you are working with a list of drinks, you could use:

```
POST  /drinks
```

Or

```
PUT /drinks/23
```

(the ID in this case is known ahead of time, ideal for an update)

So far these methods correlate to the common actions of working with data known as CRUD (create, read, update, delete).

PATCH is another method that can be used with the purpose of updating part of a resource, such as updating the description of a drink. If you're working with small objects that are in memory, you could also use PUT and send the whole object back with its new state, replacing the old.

Our Basic API Code:

```
from flask import Flask, request
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)


app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
db = SQLAlchemy(app)


class Drink(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.String(120))

    def __repr__(self):
        return f"{self.name} - {self.description}"


@app.route('/')
def index():
    return 'Hello!'


@app.route('/drinks')
def get_drinks():
    drinks = Drink.query.all()

    output = []
    for drink in drinks:
        drink_data = {'name': drink.name, 'description': drink.description}

        output.append(drink_data)

    return {"drinks": output}


@app.route('/drinks/<id>')
def get_drink(id):
    drink = Drink.query.get_or_404(id)
```

```
    return {"name": drink.name, "description": drink.description}


@app.route('/drinks', methods=['POST'])
def add_drink():
    drink = Drink(name=request.json['name'],
                  description=request.json['description'])
    db.session.add(drink)
    db.session.commit()
    return {'id': drink.id}


@app.route('/drinks/<id>', methods=['DELETE'])
def delete_drink(id):
    drink = Drink.query.get(id)
    if drink is None:
        return {"error": "not found"}
    db.session.delete(drink)
    db.session.commit()
    return {"message": "yeet!@"}
```

Extra Thoughts

Another common construct is building an SDK (software development kit).

This is a wrapper around the SDK in a particular language. So instead of making an HTTP request to get data, you can invoke a function like get_drinks which returns a list of drink objects.

This does add a layer of complexity, but could make client side development a lot easier to develop.

You should also familiarize yourself with various HTTP response codes.