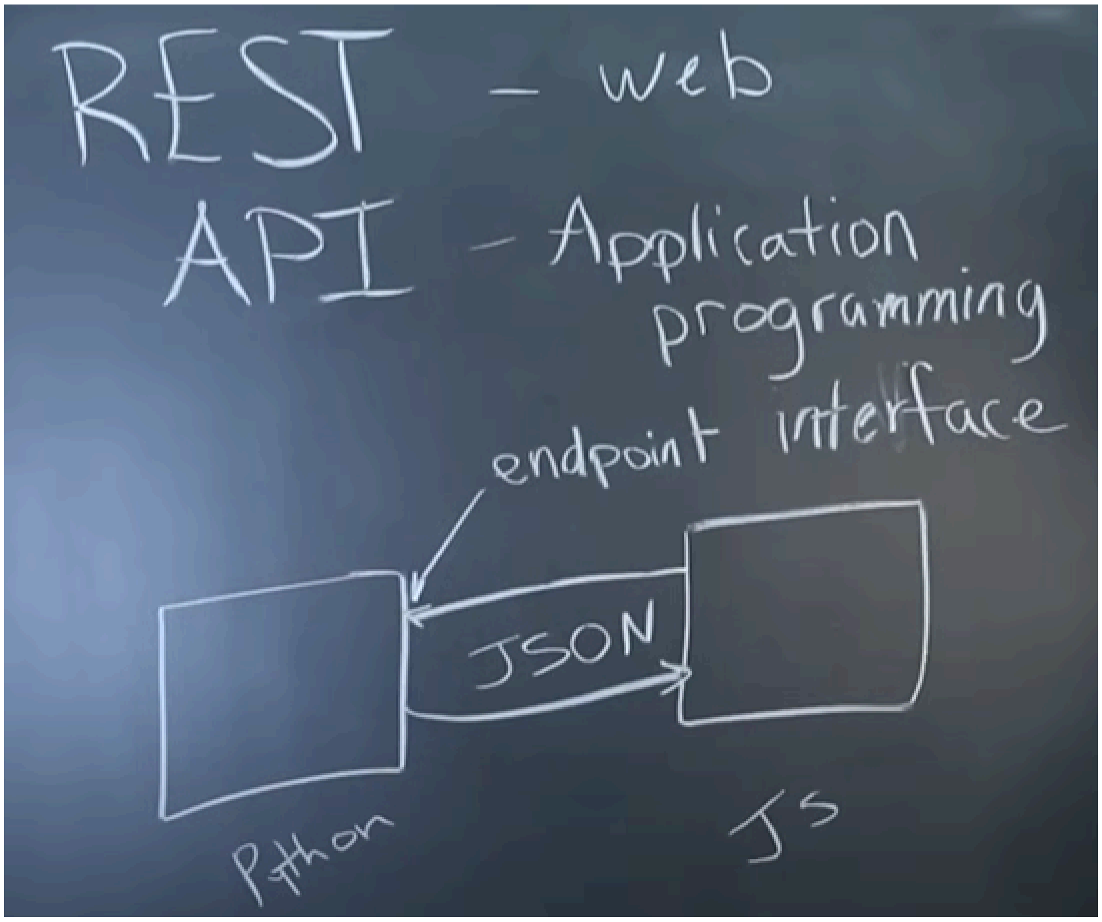


API: Application Programming Interface
An API is a set of rules and protocols that enables communication between two different software applications. It acts as a bridge, allowing one software to request data or services from another and receive a response.

In this communication:

- **Server**: The software that provides the requested data or service.
- **Client**: The software that sends the request and consumes the response.

For example:

- A Python-based application can use an API to interact with a JavaScript-based software.
- APIs ensure seamless communication, enabling the exchange of information or functionality between systems, regardless of the technologies they are built on.

REST API (Representational State Transfer API)

A **REST API** is a set of rules and conventions for building and interacting with web services. It allows communication between a client (e.g., a frontend application) and a server (e.g., a backend system) using standard web protocols, such as HTTP.

REST is like a way for two software applications to "talk" to each other over the **web or the internet**.

- **Just like humans communicate** using methods like phone calls, emails, or chats...
- **Software communicates** using REST over the web.

REST uses standard web protocols (like HTTP) to send and receive data between a client and a server.

For example:

- A client (mobile app) "talks" to a server (database) using REST to fetch or send data.

Key Principles of REST API

1. **Client-Server Architecture**:
   - The client sends requests to the server, and the server responds with the requested data or performs an action.
   - They remain independent, allowing them to evolve separately.
2. **Stateless**:
   - Each API call from the client contains all the information the server needs to process it.
   - No client context is stored on the server between requests.

What Does Stateless Mean in REST?

- **Stateless** means that every request a client sends to the server must contain all the information the server needs to fulfill it.
- The server does **not store any memory or context** about the client between requests.

Example:

1. You ask the server for user details:
   Request: `GET /users/123`
   - The server doesn't remember your last request; it just gives you the result.

2. If you need to log in or perform another task, your next request must include all necessary data (like login credentials or session tokens).

Why Stateless?

- **Scalability**: The server doesn't need to remember anything, making it easier to handle lots of requests.
- **Simplicity**: No extra server storage or logic is needed for tracking user sessions.
- **Reliability**: Each request is independent, reducing chances of errors caused by previous requests.

3. **Resource-Based**:
   - Data or functionality is treated as **resources** (e.g., `users`, `products`) and is accessed using **URLs** (Uniform Resource Locators).
   - Example:
     - `GET /users` → Fetches a list of users.
     - `POST /users` → Creates a new user.
4. **Use of HTTP Methods**:
   REST APIs use HTTP methods to perform operations:
   - **GET**: Retrieve data.
   - **POST**: Create new data.
   - **PUT**: Update existing data.
   - **DELETE**: Remove data.
5. **Representation**:
   - Resources are typically represented in formats like **JSON** or **XML**, with JSON being the most common due to its simplicity and readability.
6. **Uniform Interface**:
   - Consistency in resource naming and actions makes it easy to use. For example:
     - `/products` for all actions related to products.
     - `/orders` for orders.
7. **Stateless Caching** (Optional):
   - Responses can be cached to improve performance, provided it doesn't compromise accuracy.

Why is REST API Important?

- **Platform-independent**: Works across different systems and languages.
- **Scalable**: Follows a stateless design, making scaling easier.
- **Flexible**: Returns data in various formats, though JSON is most common.
-

What Are API Endpoints?

An **API endpoint** is like an address where a client can send requests to interact with the server. It tells the server **what you want to do** and **which data you want to work with**.

How an Endpoint Works:

1. **Base URL**: The main address of the server.
   Example: `https://api.example.com`
2. **Resource Path**: Tells the server what resource you're talking about.
   Example: `/users` or `/products`.
3. **HTTP Method**: Specifies the action you want to perform (GET, POST, etc.).

Examples of Endpoints:

1. Get all users: `GET https://api.example.com/users`
2. Get one user: `GET https://api.example.com/users/123`
3. Add a new user: `POST https://api.example.com/users`
4. Update a user: `PUT https://api.example.com/users/123`
5. Delete a user: `DELETE https://api.example.com/users/123`

Why Are Endpoints Important?

- They tell the server **exactly where to go** and **what to do**.
- Endpoints make it easy for the client to interact with the server.

Think of endpoints as the **specific addresses for getting or managing information** on the server!

Example:

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.

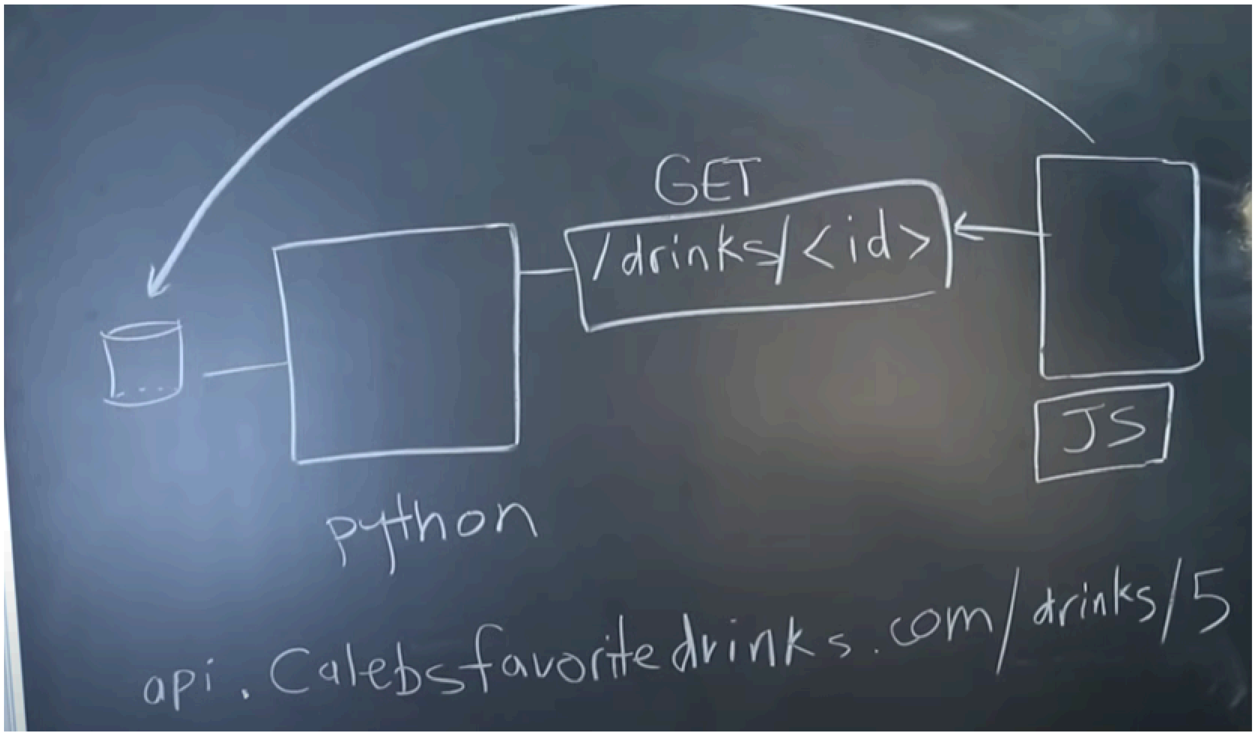
The full endpoint might be used like so: calebcurry.com/drinks/5.

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.



What's Happening Here?

1. **The App**:
   - You're building an app to store information about drinks.
   - You want to fetch details about a specific drink using its ID.
2. **Backend (Server)**:
   - The server (written in Python) connects to a database that stores the drink information.
   - When the client (e.g., a web browser or mobile app) requests a drink's details, the server fetches the info from the database and sends it back to the client.
3. **API Endpoint**:
   - The server doesn't expose everything directly. It only allows specific things to be accessed, like fetching drink details.
   - For this, you define an **endpoint**: `/drinks/<id>`
     Example: If you want details about a drink with ID `5`, you'd access:
     `api.calebsfavoritedrinks.com/drinks/5`

Why Use REST and a Server?

Instead of letting the front end directly access the database, we use a server with REST APIs. Here's why:

1. **Security**:
   - JavaScript (used for frontends) isn't secure enough to protect sensitive data.
   - A backend server adds security layers like authentication (verifying identity) and authorization (controlling access).
2. **Versatility**:
   - One backend can work with **multiple frontends** (e.g., a website, mobile app, desktop app).
   - They all share the same backend and data, making it consistent.
3. **Modularity**:
   - If we want to change the backend (e.g., switch databases), we can do so without affecting the frontend as long as the API stays the same.
4. **Interoperability**:
   - If you make your API public, **other developers** can use it to build their own apps with your data.
     Example: Developers can use your drink API to make a mobile app or a chatbot about drinks.

What's with the Slashes?

The **slashes** are part of the URL that REST APIs use to represent resources (like drinks).

- Example: `/drinks/5` means "Give me the drink with ID 5."

Summary

Think of REST as the **language** that allows the frontend (client) and backend (server) to "talk" to each other securely and efficiently over the web.

Using REST:

- The client asks the server for data through an **endpoint**.
- The server gets the data from the database and sends it back.
- This setup is **secure, modular, versatile**, and allows collaboration between developers.

Basic **HTTP methods** used in REST APIs for performing CRUD operations. Here's how you can frame it for your notes:

HTTP Methods and Their Purpose

1. **GET**
    - **Purpose**: Retrieve data from the server.
    - **Example**: Fetching a list of users or details of a specific item.
2. **POST**
    - **Purpose**: Write new data to the server (create something).
    - **Example**: Adding a new user or creating a new order.
3. **DELETE**
    - **Purpose**: Delete data (remove something).
    - **Example**: Deleting a user or removing an item from a list.
4. **PUT**
    - **Purpose**: Write to update existing data on the server.
    - **Example**: Updating a user's profile or modifying an order.