Api

Application Programming Interfaces

Define : An Application Programming Interface (API) is a contract that allows code to talk to other code

Types of APIs

Medium

While this course will focus on Web APIs, it is important to know that "API" can apply to a broad range of interfaces.

- Hardware APIs
  Interface for software to talk to hardware.
  Example: How your phone's camera talks to the operating system.
- Software Library APIs
  Interface for directly consuming code from another code base.
  Example: Using methods from a library you import into your application.
- Web APIs
  Interface for communicating across code bases over a network.
  Example: Fetching current stock prices from a finance API over the internet
- *Legacy Api*

Architecture :

There is more than one way to build and consume APIs. Some architecture types you may come across are:

- REST (Representational State Transfer)
- GraphQL
- WebSockets
- webhooks
- SOAP (Simple Object Access Protocol)
- gRPC (Google Remote Procedure Call)
- MQTT (MQ Telemetry Transport

REST APIs
Some traits of REST APIs include not storing session state between requests, the ability to cache, and the ability to send and receive various data types. Still confused? Don't worry; we will learn hands-on very soon in this course!

Access

APIs also vary in the scope of who can access them.

- **Public APIs (aka Open APIs)**
  Consumed by anyone who discovers the API

- **Private APIs**
  Consumed only within an organization and not made public

- **Partner APIs**
  Consumed between one or more organizations that have an established relationship

REST

Related: REST API client in Postman

REST APIs are designed to make server-side data readily available by representing it in simple formats such as JSON and XML. The acronym stands for REpresentational State Transfer, and it was released in 2000 after being introduced in an academic thesis by Roy Fielding. This particular type of API adheres to six specific architectural constraints:

- A uniform interface
- Completely stateless
- Native caching
- Client-server architecture
- A layered system
- The ability to provide executable code to the client

Pros

Operations are executed with different HTTP methods including GET, POST, PUT, DELETE, OPTIONS, and PATCH. By leveraging these functions, REST APIs become extremely capable across the internet.

The key benefits of this API type are that the client and the server are completely decoupled from one another. This allows for abstraction layers that help to maintain flexibility even as a system grows and evolves. In addition, REST is cache friendly and supports multiple formats, which is important when you're building public-facing APIs. The flexible data formatting that you get as a result makes them extremely useful for varied applications.

REST APIs are most commonly used as management APIs to interact with objects in a system. They also are useful when you're building simple resource-driven apps that don't need much in terms of query flexibility.

Cons

Where REST APIs fall short is that their rich metadata creates big payloads that can sometimes cause more trouble than they're worth. You can get over- and under-fetching problems that require further API requests, bogging down the process.

From an industry perspective, another key externality resulting from the flexibility is that there is no binding contract on what structure is used for messages. As a result, there is a lot of back and forth when it comes to implementation – which can cause unnecessary frustration and bottlenecks.

Request methods

When we make an HTTP call to a server, we specify a **request method** that indicates the type of operation we are about to perform. These are also called **HTTP verbs**.

Some common HTTP request methods correspond to the CRUD operations mentioned earlier. You can see a list of more methods here.

Method name	Operation
GET	Retrieve data (**R**ead)
POST	Send data (**C**reate)
PUT/PATCH	Update data (**U**pdate) * **PUT** usually replaces an entire resource, whereas **PATCH** usually is for partial updates
DELETE	Delete data (**D**elete)

Protocol : host : path

Protocol : http

Host : library-api.postman.com

Path : /books

Request URL

In addition to a request method, a request must include a **request URL** that indicates *where* to make the API call. A request URL has three parts: a **protocol** (such as **http://** or **https://**), **host** (location of the server), and **path** (route on the server). In REST APIs, the path often points to a reference entity, like "books".

Paths and complete URLs are also sometimes called **API endpoints**.

Response status codes

The Postman Library API v2 has returned a response status code of "200 OK". Status codes are indicators of whether a request failed or succeeded.

Status codes have conventions. For example, any status code starting with a "2xx" (a "200-level response") represents a successful call. Get familiar with other status code categories:

Code range	Meaning	Example
2xx	Success	200 - OK
		201 - Created
		204 - No content (silent OK)
3xx	Redirection	301 - Moved (path changed)
4xx	Client error	400 - Bad request
		401 - Unauthorized
		403 - Not Permitted
		404 - Not Found
5xx	Server error	500 - Internal server error
		502 - Bad gateway
		504 - Gateway timeout

Variables in Postman

Postman allows you to save values as variables to reuse them and easily hide sensitive information like API Keys.

We will use a variable to replace our base URL so that we don't have to type that repeatedly. Once a variable is defined, you can access its value using double curly brace syntax like this: `{{variableName}}`

Query parameters

Remember that the minimum ingredients you need to make a request are:

- a request method (`GET`/`POST`/`PUT`/`PATCH`/`DELETE`, etc)
- a request URL

Some APIs allow you to refine your request further with key-value pairs called **query parameters**.

Query parameter syntax

Query parameters are added to the end of the path. They start with a question mark `?`, followed by the key-value pairs in the format: `<key>=<value>`. For example, this request might fetch all photos that have landscape orientation:

```
GET https://some-api.com/photos?orientation=landscape
```

If there are multiple query parameters, each is separated by an ampersand `&`. Below two query parameters to specify the orientation and size of the photos to be returned:

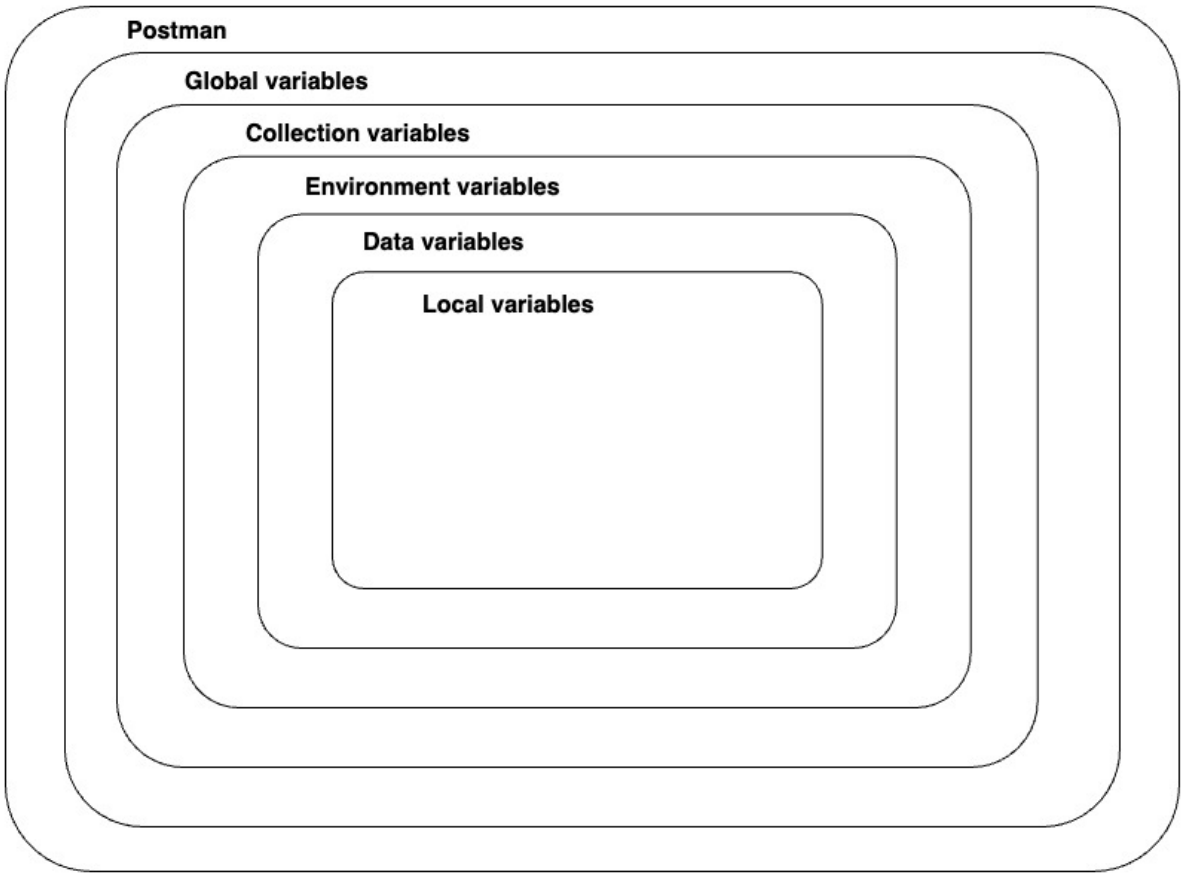
```
GET https://some-api.com/photos?orientation=landscape&size=500x400
```

`post method :`

```
add api-key and value in header portion
```

```
its authorized that’s why we need api key to access it end point for post
```

```
delete the data
```



Scripting in Postman

Postman allows you to add automation and dynamic behaviors to your collections with scripting.

Postman will automatically execute any provided scripts during two events in the request flow:

1. Immediately before a request is sent: pre-request script (**Pre-request Script** of Scripts tab).

2. Immediately after a response comes back: post-response script (**Post-response** of Scripts tab).

In this lesson, we will focus on writing scripts in the **Post-response** tab, which are executed when a response comes back from an API.

The `pm` object

Postman has a helper object named `pm` that gives you access to data about your Postman environment, requests, responses, variables and testing utilities.

For example, you can access the JSON response body from an API with:

```
pm.response.json()
```

You can also programmatically get collection variables like the value of `baseUrl` with:

```
pm.collectionVariables.get(“baseUrl”)
```

In addition to getting variables, you can also set them with `pm.collectionVariables.set("variableName", "variableValue")` like this:

```
pm.collectionVariables.set(“myVar”, “foo”)
```