Postman API

Fundamentals



**Learning Objectives**

* What APIs are and why they are crucial to modern software development.
* How to use Post to work with APIs.
* How to interact with a real-world API.
* How Postman helps you benefit from the power of APIs in your applications.

**Application Programming Interfaces**

Its a contract that allows code to talk to other code.

APIs are the building blocks o modern software because they allow for sharing of resources and services across applications, organizations, and devices.

**Importance of APIs**

* Thy assist developers **integrate exciting feature,** and **build automation** without reinventing the wheel.
* They allow enterprises **open up their product for faster innovation.**
* They can **be products themselves.**
* They separate the **requesting application from the infrastructure** of the responding service and offer layers of security between the two as they communicate.
* They can also provide **another** layer **of protection for personal users.**

**Types of APIs**

**Hardware APIs :**

Interface for software to talk to hardware. ( An app to talk to your camera )

**Software Library APIs :**

Interface for directly consuming code from another code base. ( Image to be processed with filters )

**Web APIs :**

Interface for communicating across code bases over a network. ( Send an image to instagrams’ servers so that friends can view it )

**Architectures**

There is more than one way to build and consume APIs. These are :

1. REST ( Representational State Transfer ).
2. GraphQL.
3. Web Sockets.
4. Web Hooks.
5. SOAP ( Simple Object Access Protocol ).
6. gRPC (Google Remote Procedure Call ).
7. MQTT ( MQ Telemetry Transport ).

**1. REST ( Representational State Transfer )**

They are designed to make server-side data readily available by presenting it in simple formats such as **JSON** and **XML.** 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 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.

Its cache friendly and supports multiple formats, which is important when you’re building public-facing APIs. The data formatting makes them extremely useful for varied applications. They are mostly used as management APIs to interact with objects in a system and when building simple resource-driven apps that don’t need much in terms of query flexibility.

**Cons**

Their rich metadata create big payloads that can sometimes cause more trouble. You can get over- and under-fetching problems that require further API requests, bogging down the process.

There are 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.

**2. SOAP ( Simple Object Access Protocol )**

SOAP API are formatted as XML files and they are extremely common web communication protocols.

**Pros**

Its completely agnostic when it comes to programming language and processing platform. The standardized format ensures that no matter what is receiving the message on the other end, the request can be executed.

It comes with native error handling already built-in, helping developers to be proactive and solve issues before they snowball.

With high-security data transmissions in situations where two parties have agreed to a specific legal contract. SOAP standardization works wonders because it allows for the contract’s terms to be formally codified and enforced throughout all of APIs processing.

**Cons**

SOAP’s standardization makes requests incredibly accessible to applications but as a side effect, the format can become very formal and verbose. Every message must include an envelope tag at the start and at the end, a body that incudes the actual request, header for specific information and additional requirements, as well as any faults that occur throughout processing.

SOAP requires a sheer volume of information. The XML files are large and are often unnecessary clunky – especially for simple systems.

The number of people who specialize in SOAP servers is rapidly declining, and that makes it difficult thing to maintain.

**3. GraphQL**

It was developed in response to REST APIs, with the idea that you could execute precise syntax that retrieves only what is needed., lightening the payload and simplifying the process significantly.

Its an internal Facebook protocol that was first released in 2015. You can start by creating a schema that describes all the possible queries and the specific types that they return. This can be challenging, but once completed, the API can accept specific requests and return a result that matches exactly what the user is looking for.

**Pros**

API request are transparent and well documented, giving users all the information they need to use it effectively. The precise results, detailed error messages, and flexible permissions help round out a well-balanced but high-functioning API.

When it comes to data structuring, GraphQL gives users significant flexibility.

**Cons**

Too many nested fields in any one request can cause the performance to suffer.

It requires custom efforts to achieve proper caching, it doesn’t reuse standard HTTP caching semantics. It difficult to pickup without lots of training and experience.

**3. gRPC**

In gRPC, RPC stands for Remote Procedure Call and refers to something that can execute a function housed elsewhere but in a different context. The *‘g’* appended at the beginning narrows down to the most advanced version developed by Google back in 2015.

A user on one side will select a remote procedure o execute, serialize the necessary parameters and then append any additional information into the message. This will then be sent to the server, which interacts with other application, decoding the message and executing the operation. A result then comes back to the initial user.

**Pros**

Its simplicity, straightforward makes it powerful, using GET to fetch information and POST for everything else. Functions are easy to add, and for lightweight payloads, you get great performance overall. The ability to define any type of function.

Command APIs that send simple requests to remote systems and customer-specific APIs, help manage internal micro-services at scale with great speed.

It has become a staple of the Docker-based application world, proving its value when there is massive numbers of remote calls to execute through simplifying otherwise complex remote calls.

**Cons**

It is tightly coupled to the underlying system, which restricts its re-usability in many cases. There is no abstraction layer sitting between the API and the actual system functions, which can raise security concerns.

**3. Web-sockets**

Web-sockets is a communications protocol that provide dynamic communication over a single TCP connection. There is an persistent connection between the client and the server and both parties can start sending data.

It enables bidirectional, full-duplex communication.

**Pros**

It acts as a thin transport layer that sits on top of TCP/IP technology stack. It can send message to a server and receive event-driven responses without needing to poll the server fo a reply.

Its an event-driven technology, data is pushed as soon as it becomes available.

**Cons**

Web-sockets is based on HTTP/1.1 where as gRPC was built using HTTP/2.

Web-sockets don’t automatically recover when connections are terminated.

Its stateful, hard to use in large-scale systems that consist of multiple Web-sockets servers.

**Access**

**Public 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.

**Introducing Postman**

**An API Platform**

**Postman** is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so as to create better APIs and faster and consume them with ease.

**Working with API then and now:**

**cURL vs Postman**

Before Postman, it was common practice to poke at APIs with a command line tool for making HTTP requests called **cURL.**

**cURL** works great , but once you make the call, the API response data is lost in the river of the terminal. With no visibility of the metadata of the response without adding more details to the command.

**Postman** shows the response with clean indents and colors and allows to save, organize and share the requests. All the components of the requests and response broken down into tabs and other helpful details like the response time and status code are visible.

**Request-Response Pattern**

Request-Response pattern represents how computers communicate over a network. An API is the interface that lets us know what kind of response to expect when we make certain calls to a server.

The **client** is the agent making a request. A client could be a browser or an application.

The **request** is sent over a **network** to some **server.** The server interprets the request and sends the appropriate **response** over the network back to the Postman Client.

**Request Parameters**

**Variables in Postman**

Postman allows saving values as **variables** to reuse them and easily hide sensitive information like API keys.

Variables are used to replace the base URL to avoid typing repeatedly. Once a variable is defined, its values can be accessed using double curly brace syntax: **{{variableName}}**

There ae two columns:

* **Initial Value –** the value initially set when someone forks or imports the collection. **NOTE:** If the collection is shared with others, they will see this value, so don’t put any secrets here!.
* **Current Value –** Postman always resolves the variable to this value. This is local to a Postman account, and not public. Its good to keep secrets like API keys only in this column and not include them in the initial value column.

**Query Parameters**

The minimum ingredients required to make a request are:

* a request method (GET/POST/PUT/PATCH/DELETE)
* a request URL

Some APIs allow refining requests 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:

**GET** [**https://some-api.com/photos?orientation=landscape**](https://some-api.com/phottos?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=500\*400**

**When to use query parameters**

: Read the API Documentation.

Sometimes, query parameters are optional and allow you to add filters or extra data to your responses. Sometimes, they are required in order for the server to process y=the requests. APIs are implemented differently to fulfill different needs.

**Path Variable**

Another way of passing request data to an API is via **path variables.** A path variable is a dynamic section of a path and is often used for IDs and entity names such as usernames.

**Path Variable syntax**

The path variable comes immediately after a slash in the path. Example:

**GET** [**https://api.github.com/users/**](https://api.github.com/users/)**{username} –** Making this API call with a value for {username} will fetch data about that user.

It can have multiple path variables in a single request, such as this endpoint for getting a user’s GitHub code repository.

**GET** [**https://api.github.com/repos/**](https://api.github.com/repos/)**{owner}/{repoName}**

**Path vs Query Parameters**

|  |  |
| --- | --- |
| **Path Variable** | **Query Parameters** |
| Ex: /books/abc123 | Ex: /books?search=borges&checkedOut=false |
| Located **directly after a slash** in the path.  It can be **anywhere on the path.** | Located only at the **end of a path**, right after a question mark **?.** |
| Accepts **dynamic values** | Accepts **defined query keys with potential dynamic values.** |
| Often used for IDs or entity names | Often used for options and filters. |

***\**** *These are just conventions! Some APIs might ask you to pass an ID or username in a query parameter like this:* ***/users?username=getpostman.***

**When to use Path Variable?**

**Always read the API documentation!** If a path parameter is required, the documentation will mention this.

**NOTE:** Some API documentation uses colon syntax to represent a wildcard in the path like **/users/:username**, while some use curly braces like **/users/{username}**. They both mean the same thing: that part of the path is dynamic.

**Authorization**

API which have completely open endpoints can be accessed by anyone publicly. It would allow unauthorized people to access data they should not see, or allow bots to flood an API with thousands of calls per second and shut it down.

There are multiple methods for authorizing a request. Example:

* **Basic Auth** (username and password).
* **OAuth** (delegated authorization).
* **API Keys** (secret strings registered to a developer from an API portal).

**Getting an API Key**

APIs that use API Key auth usually allow developers to sign up in a developer portal, where they will receive a random API Key that can be used to authorize their requests to the API. The API Key allows the API to track who is making calls and how often.

**Variables in Postman (Cont’d)**

**Headers**

Headers are how to add **metadata** about our requests, such as authorization information or specify the data type requested in the response. This is different than the actual payload data sent in the body of a request.

Headers are like the outside of an envelope when sending a letter. The envelope has information about delivering the letter, like proof that the postage is paid. The actual data “payload” is the letter inside the envelope.

Using variables saves time and help reduce redundant cop-paste of the request URL using double curly braces.

Postman allows you to save values as variables so that :

1. Reuse values to keep your work DRY (Don’t Repeat Yourself).
2. Hide sensitive values like API keys from being shared publicly.

**Variable Scopes**

Variables can be set that live at various scopes. Postman will resolve to the value at the nearest and narrowest scope.

From broadest to narrowest, this scopes are **global, collection, environment, data,** and **local.**

* *If a variable with the same name is declared in two different scopes, the value stored in the variable with the narrowest scope will be used. For example: if there is a global variable named* ***username*** *and a local variable named* ***username,*** *the local value will be used when the request runs.*

**Scripting in Postman**

Postman allows for adding of automation and dynamic behaviors to collections with scripting and will automatically execute any provided scripts during two events in the request flow:

1. Immediately before a script request is sent: pre-request script (**Pre-request Script** of script tab).
2. Immediately after a response comes back: post-response (**Post-response** of script tab).

**The ‘pm’ object**

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

* *Access the JSON response body from an API with:* 
  + - * ***pm.response.json***

Its also possible to programmatically get collection variables like the value of **baseUrl** with:

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

Its can also set them with ***pm.collectionVariables.set(“variableName”, “variableValue”)*** like this:

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