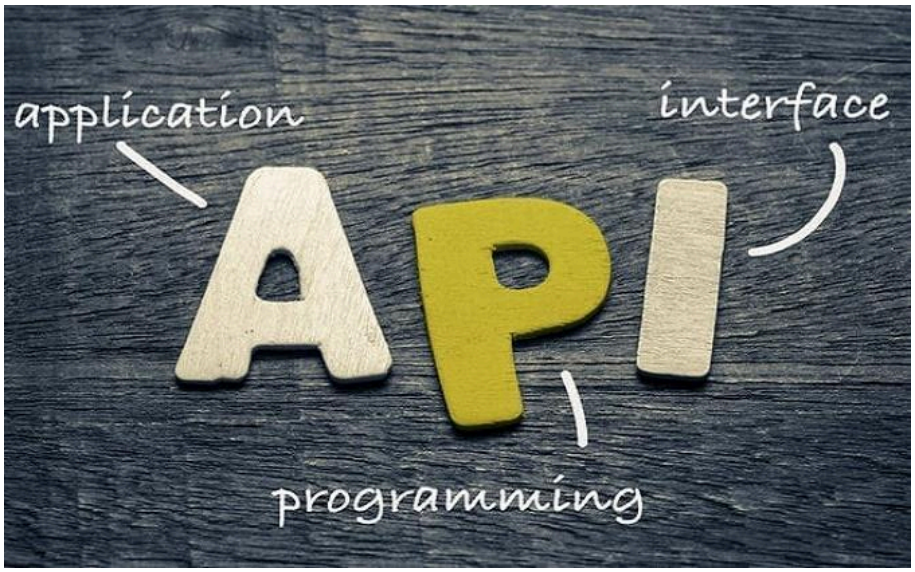
API Basics and API Testing!!

This document will provide you with an overview of API and the API Testing process. It details the types of API testing and its protocols, various HTTP methods, and response codes.

Pricilla B
bpricilla@gmail.com
https://www.linkedin.com/in/pricilla-b/

Table of Contents

What is an API?



API is an acronym and it stands for **A**pplication **P**rogramming **I**nterface. API is a set of routines, protocols, and tools for building Software Applications. APIs specify how one software program should interact with other software programs.

Normally, API facilitates the reusability instead of developing something new. For example

1. *I need to book a room in Hyatt Regency via Agoda. I can directly do it on the Hyatt Regency website, or on travel booking websites like Agoda, Trivago, etc. So here the Hyatt Regency develops an API and provides specific(read/write) access to the travel agencies via which they can view/book their hotels.*

2. *Well instead of having to create Google Maps all over again from scratch, you can connect your app to Google Maps. And how do you do that? Connect it to Google Maps API.*

Private APIs:

Internal APIs are the opposite of open APIs in that they are inaccessible to external consumers and only available to an organization’s internal developers. Internal APIs can enable enterprise-wide initiatives from the adoption of DevOps and microservice architectures to legacy modernization and digital transformation. The use and reuse of these APIs can enhance an organization's productivity, efficiency, and agility.

An example of a reusable internal API is if a call center team created a customer information API used in a call center application to access their name, contact information, account info, etc. That team can then reuse this same API in a customer-facing web application or mobile application.

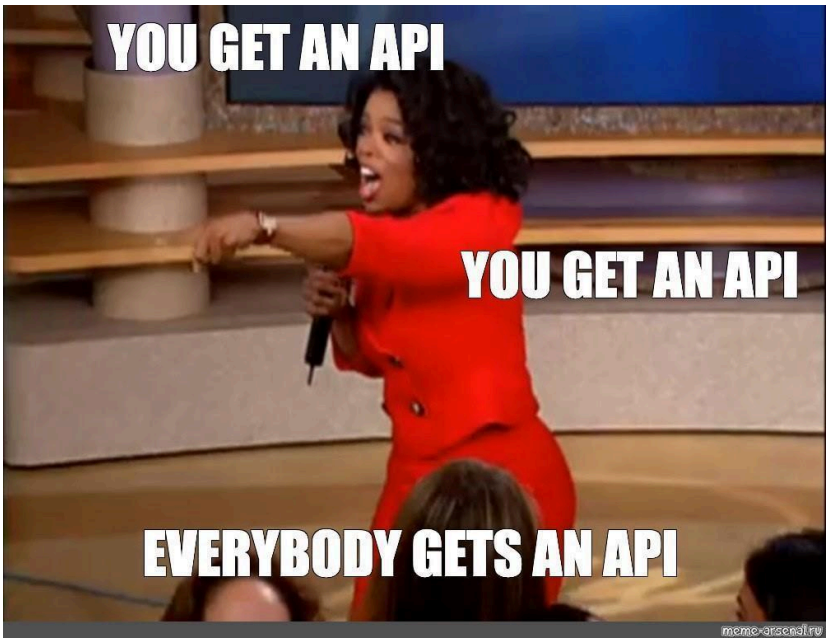
Public/Open APIs:

Open APIs, on the other hand, provide external developers with easy access and integrate information from one tool to another. An open or public API saves developers time by allowing them to connect their platform with previously existing tools, reducing the need to create entirely new functions.
Eg: Google Maps

Partner APIs:

Partner APIs fall somewhere in the middle of internal and external APIs. They are APIs that are accessed by others outside the organization with exclusive permissions. Usually, this special access is afforded to specific third parties to facilitate a strategic business partnership.

A common use case of a partner API is when two organizations want to share data with each other — such as a county's health department and a hospital within that county. A partner API would be set up so each organization has access to the necessary data with the right set of credentials and permissions.



Client Service Architecture:

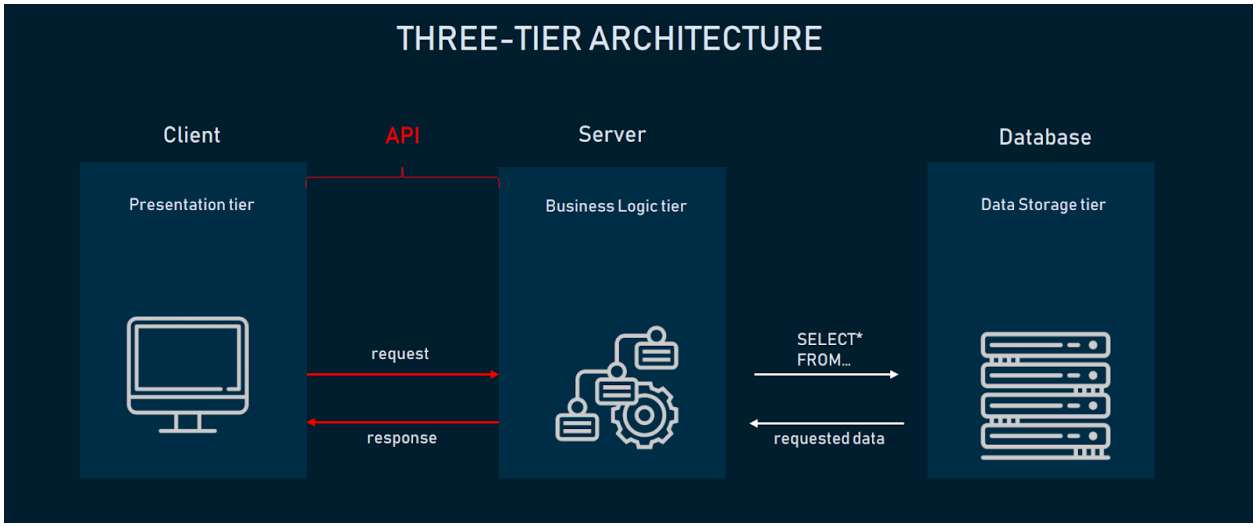
The client-server model, or client-server architecture, is a distributed application framework dividing tasks between servers and clients, which either reside in the same system or communicate through a computer network or the Internet. The client relies on sending a request to another program in order to access a service made available by a server. The server runs one or more programs that share resources with and distribute work among clients.

The client-server relationship communicates in a request-response messaging pattern and must adhere to a common communications protocol, which formally defines the rules, language, and dialog patterns to be used. Client-server communication typically adheres to the TCP/IP protocol suite.

TCP protocol maintains a connection until the client and server have completed the message exchange. TCP protocol determines the best way to distribute application data into packets that networks can deliver, transfers packets to and receives packets from the network, and manages flow control and retransmission of dropped or garbled packets. IP is a connectionless protocol in which each packet traveling through the Internet is an independent unit of data unrelated to any other data units.
Client requests are organized and prioritized in a scheduling system, which helps servers cope with the instance of receiving requests from many distinct clients in a short space of time. The client-server approach enables any general-purpose computer to expand its capabilities by utilizing the shared resources of other hosts. Popular client-server applications include email, the World Wide Web, and network printing.

There are four main categories of client-server computing:
One-Tier architecture: consists of a simple program running on a single computer without requiring access to the network. User requests don't manage any network protocols, therefore the code is simple and the network is relieved of the extra traffic.
Two-Tier architecture: consists of the client, the server, and the protocol that links the two tiers. The Graphical User Interface code resides on the client host and the domain logic resides on the server host. The client-server GUI is written in high-level languages such as C++ and Java.
Three-Tier architecture: consists of a presentation tier, which is the User Interface layer, the application tier, which is the service layer that performs detailed processing, and the data tier, which consists of a database server that stores information.

N-Tier architecture: divides an application into logical layers, which separate responsibilities and manage dependencies, and physical tiers, which run on separate machines, improve scalability and add latency from the additional network communication. N-Tier architecture can be closed-layer, in which a layer can only communicate with the next layer down, or open-layer, in which a layer can communicate with any layers below it.



What is API Testing?

Testing an API is as simple as submitting a request on behalf of an application (Client), using another application's API (Server), and checking that it returns the expected response.
API testing is about testing the APIs directly and also as a part of integration testing to check whether the API meets expectations in terms of functionality, reliability, performance, and security of an application. In API Testing our main focus will be on a Business logic layer of the software architecture. API testing can be performed on any software system which contains multiple APIs. API testing won't concentrate on the look and feel of the application. API testing is entirely different from GUI Testing.

How is UI testing different from API testing?

UI (User Interface) testing is to test the graphical interface part of the application. Its main focus is to test the look and feel of an application. On the other hand, API testing enables the communication between two different software systems. Its main focus is on the business layer of the application.

API testing is one of the most challenging parts of the chain of software and QA testing because it works to assure that our digital lives run in an increasingly seamless and efficient manner.

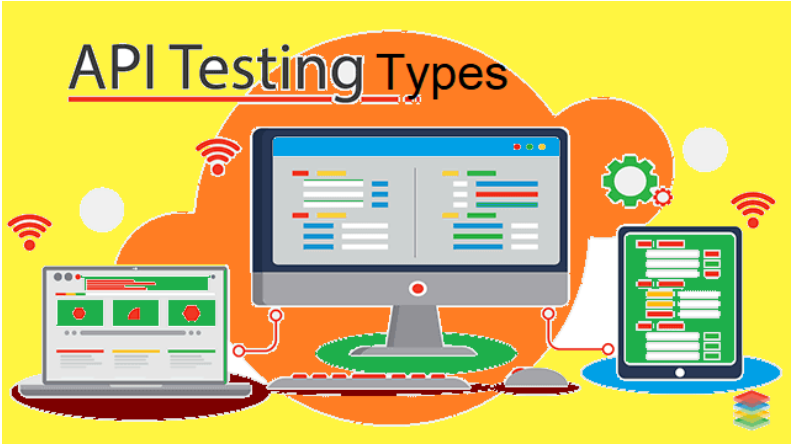
While developers tend to test only the functionalities they are working on, testers are in charge of testing both individual functionalities and a series or chain of functionalities, discovering how they work together from end to end.

APIs are what give value to an application. It's what makes our phones "smart", and it's what streamlines business processes. If an API doesn't work efficiently and effectively, it will never be adopted, regardless if it is free or not. Also, if an API breaks because errors weren't detected, there is the threat of not only breaking a single application, but an entire chain of business processes hinged to it.

Why API Testing?

- API Testing is time effective when compared to GUI Testing. API test automation requires less code so it can provide faster and better test coverage
- API Testing helps us to reduce the testing cost(shift-left defects). With API Testing we can find minor bugs before the GUI Testing. These minor bugs will become bigger during GUI Testing
- API Testing is language independent
- API Testing is quite helpful in testing Core Functionality
- API Testing helps us to foresee and reduce the risks of the system

Types of API Testing:



Validation:

Validation testing is done immediately following the development process, specifically after verification of the API's constituent parts and functions is completed. Validation testing is essentially a set of simple questions applied to the entirety of the project. These questions include:

Schema: Is the API following the correct schema?
Ultimately, this test can be simply said to be an assurance of correct development against the stated user needs and requirements.

Product: Did we build the correct product? Is the API itself the correct product for the issue that was provided, and did the API experience any significant code bloat or feature creep that took an otherwise lean and focused implementation into an untenable direction?

Behavior: Is the API accessing the correct data in a correctly defined manner? Is the API accessing too much data, is it storing this data correctly given the confidentiality and integrity requirements of the dataset?

Efficiency: Is the API the most accurate, optimized, and efficient method of doing what is required? Can any codebase be removed or altered to remove impairments to the general service?

Functional Testing:

The purpose of functional testing is to ensure that you can send a request and get back the anticipated response along with the status. This includes negative and positive testing. Make sure to cover all of the possible data combinations.

There could be bugs rooted in the unit level or backend that wouldn't be visible via UI testing. Error handling scenarios that are not feasible via the front end can be covered by API testing.

Functional testing is still a very broad methodology of testing but is less broad than those under Validation testing. Functional testing is simply a test of specific functions within the codebase. These functions in turn represent specific scenarios to ensure that the API functions within expected parameters, and that errors are handled well when the results are outside of the expected parameters.

Security Testing:

The purpose of the security testing is to make sure that the communication with the API is secure and that only the authorized user is allowed to make calls/access the API.

Need to check on the below points as part of security testing:

- To verify if the data is encrypted appropriately
- Type of Authentication used
- If tokens are used, then you need to test the validity of the token

Load Testing:

The purpose of performance testing is to ensure that the API can handle user load and determine what happens when it reaches that load limit.

Increase the number of API calls and then monitor response times and throughput.
Monitor for memory leaks by performing an endurance test. Stress the system out by loading it down with calls – how does it respond to failures and breakage.
Load testing is thus typically done after the completion of a specific unit or the codebase as a whole, testing whether the theoretical solution works as a practical solution under a given load.

Integration Testing:

The purpose of integration testing is to verify success where multiple APIs are working together. Focus on call sequencing and ensure data is returned promptly and accurately.

Documentation Testing:

The purpose of reviewing documentation is to validate that the documentation provides enough information to interact with the API. This is typically done as you're testing the API.

Do you have the information you need to successfully execute the testing? Would someone consuming this information know how to interact with the API?
Ref: https://stripe.com/docs/api

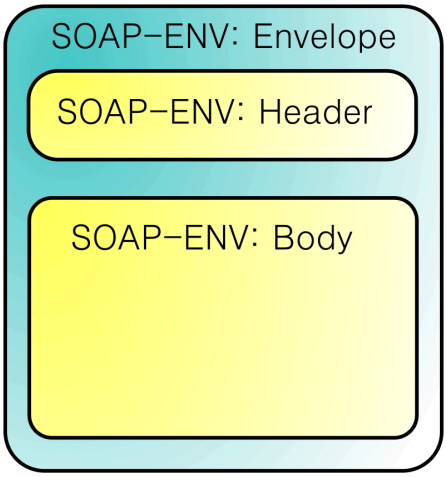
Types of API Protocols:

While there are multiple types of API protocols, the most commonly used ones are SOAP and REST.

SOAP:

SOAP is an XML-based protocol for accessing web services over HTTP. It has some specifications which could be used across all applications.

SOAP is known as the Simple Object Access Protocol, but in later times was just shortened to SOAP v1.2. SOAP is a protocol or in other words, is a definition of how web services talk to each other or talk to client applications that invoke them. The diagram below shows the various building blocks of a SOAP Message.



The SOAP message is nothing but a mere XML document that has the below components.

- An Envelope element that identifies the XML document as a SOAP message – This is the containing part of the SOAP message and is used to encapsulate all the details in the SOAP message. This is the root element in the SOAP message.
- A Header element that contains header information – The header element can contain information such as authentication credentials which can be used by the calling application. It can also contain the definition of complex types which could be used in the SOAP message. By default, the SOAP message can contain parameters which could be of simple types such as strings and numbers, but can also be a complex object types.
- A Body element that contains call and response information – This element is what contains the actual data which needs to be sent between the web service and the calling application. Below is an example of the SOAP body which actually works on the complex type defined in the header section. Here is the response of the Tutorial Name and Tutorial Description that is sent to the calling application which calls this web service.

SOAP Examples:

Request:

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <NumberToWords xmlns="http://www.dataaccess.com/webservicesserver/">
      <ubiNum>500</ubiNum>
    </NumberToWords>
  </soap:Body>
</soap:Envelope>
```

Response:

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <m:NumberToWordsResponse xmlns:m="http://www.dataaccess.com/webservicesserver/">
            <m:NumberToWordsResult>five hundred </m:NumberToWordsResult>
        </m:NumberToWordsResponse>
    </soap:Body>
</soap:Envelope>
```

REST:

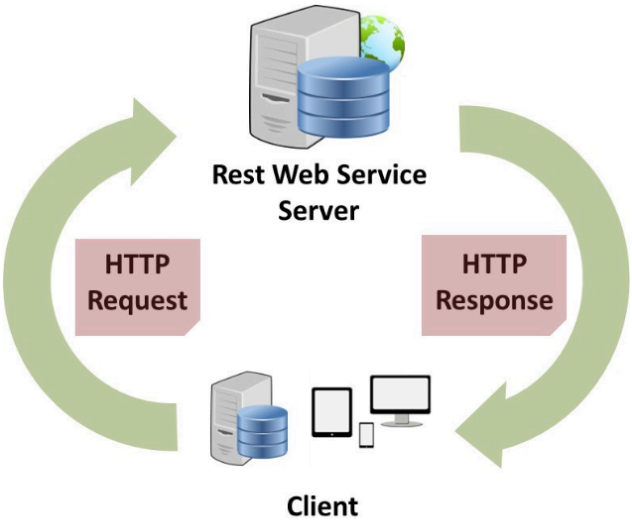
REST (**Representational State Transfer**) is an architectural style for providing standards between computer systems on the web, making it easier for **systems to communicate with each other**. REST-compliant systems, often called **RESTful** systems, are characterized by how they are stateless and separate the concerns of client and server. Most commonly used nowadays.

Separation of Client and Server

In the REST architectural style, the implementation of the client and the implementation of the server can be done **independently without each knowing about the other**. This means that the code on the client-side can be changed at any time **without affecting** the operation of the server, and the code on the server-side can be changed without affecting the operation of the client.

As long as each side **knows what format of messages to send to the other**, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we **improve the flexibility of the interface** across platforms and improve scalability by simplifying the server components. Additionally, the separation allows each **component the ability to evolve independently.**

By using a REST interface, different clients hit the same **REST endpoints**, perform the same actions, and receive the same responses.



Statelessness

Systems that follow the **REST paradigm are stateless,** meaning that the server does not need to **know anything about what state the client is in and vice versa**. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of resources, rather than commands. **Resources are the nouns of the Web** - they describe any object, document, or thing that you may need to store or send to other services. Because REST systems interact through standard operations on resources, they **do not rely on the implementation of interfaces**.

These constraints help RESTful applications achieve reliability, quick performance, and scalability, as components that can be managed, updated, and reused without affecting the system as a whole, even during the operation of the system.

Communication between client and server

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests.

REST Examples:

Request in JSON format:

```
{
"match": "Cup Final",
"against": "Academical"
}
```

Response in JSON format:

To know about the SOAP and REST differences please check here.

JSON:

JSON (JavaScript Object Notation) is an **open-standard file format** or data interchange format that uses **human-readable** text to transmit data objects consisting of **attribute - value pairs** and **array data types** (or any other serializable value). It is a very common data format, with a diverse range of applications, such as serving as a replacement for **XML in AJAX** systems.

JSON is a **language-independent** data format. It was derived from JavaScript, but many modern **programming languages** include code to generate and parse JSON-format data. The official Internet **media type** for JSON is *application/json*. JSON file names use the extension *.json* .

A JSON object contains data in the form **of a key/value pair**. The keys are strings and the values are the JSON types. Keys and values are separated by a colon. Each entry (key/value pair) is separated by a **comma**. The **{** (curly brace) represents the JSON object. An example of JSON is provided below.

```
{
   "book":[
      {
         "id":"444",
         "language":"C",
         "edition":"First",
         "author":"Dennis Ritchie "
      },
      {
         "id":"555",
         "language":"C++",
         "edition":"second",
         "author":" Bjarne Stroustrup "
      }
   ]
}
```

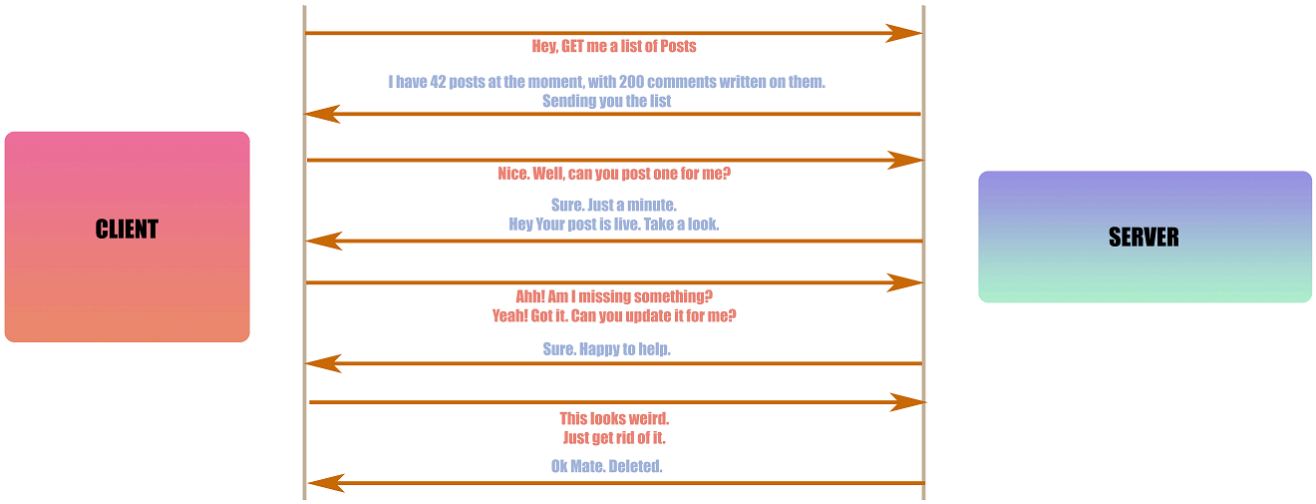
Important benefits of using JSON:

- Provide support for all browsers
- Easy to read and write
- Straightforward syntax
- You can natively parse in JavaScript using eval() function
- Easy to create and manipulate
- Supported by all major JavaScript frameworks
- Supported by most backend technologies
- JSON is recognized natively by JavaScript
- It allows you to transmit and serialize structured data using a network connection
- You can use it with modern programming languages
- JSON is text which can be converted to any object of JavaScript into JSON and send this JSON to the server

HTTP methods:

HTTP	POST	GET	PUT	DELETE
SQL	INSERT	SELECT	UPDATE	DELETE
CRUD	CREATE	READ	UPDATE	DELETE

The HyperText Transfer Protocol – HTTP – defines several methods (referred to as "verbs") that indicate the desired action to be performed on a resource. The resource is specified by the URI (Uniform Resource Identifier), more commonly, the URL. This resource may be pre-existing data or data that is generated dynamically, it depends on the server implementation. The server can be configured to support any combination of methods.
The most common are: ***GET***, ***POST***, ***PUT***, and ***DELETE***.
The concept of *idempotence* is relevant to this discussion. If something is idempotent, then no matter how many times you do it, the result will always be the same.



SOAP APIs, when sending over HTTP, can use only the POST verb, and the exact action depends on the SOAP method that is being called. REST, being an architectural style and not a standard, makes full use of all the available verbs. There is no definite answer to exactly what each verb should or should not do. When testing a RESTful API, you can use the following best practices as a starting point, but check with your in-house architect or development lead to find out what exactly your project adheres to.

GET

The GET operation is normally used to only retrieve information from the system. Nothing is added or changed, so it is more than idempotent, it is actually *nullipotent* – it has absolutely no side-effect on the data, other than possibly logging. While sending the request usually some parameters are sent along with the URL to fetch the particular record.

Eg: https://postman-echo.com/get?foo1=bar1&foo2=bar2

Generally, the body is empty which implies nothing is sent to update.

POST

POST is the only method that is assumed to be non-idempotent out of four methods. This is the preferred method when creating new objects in an application, for example creating a new order. Every POST method call should result in a new object being created (or possibly deleted) in the database.
Here we will be passing the data into the request body in different formats as per the API design.

Eg: postman-api-learner.glitch.me/info

And in the body it is mentioned as

```
{
   "name": "Ryan"
}
```

PUT

The PUT method should be idempotent. The word “should” indicates that the server is able to implement this method differently. A tester should flag such an implementation as an inconsistency.
PUT can still be used for creating objects, although since it is idempotent, repeatedly executing the same request will have the same end result as the first time. If the Request-URI refers to an already existing resource – an update operation will happen, otherwise, the create operation should happen if Request-URI is a valid resource URI.

Eg: postman-api-learner.glitch.me/info?id=2324

And in the body it is mentioned as

```
{
   "name": "Ian"
}
```

DELETE

The DELETE method is idempotent; multiple requests should result in only one thing being deleted. As an example, consider the above scenario where multiple POST requests were sent to the server for a new order, resulting in multiple orders of the same product. A DELETE request should accept a unique identifier to remove only one of the products from the order, thereby sending the same DELETE request will result in the correct idempotent operation: the one instance of the product.

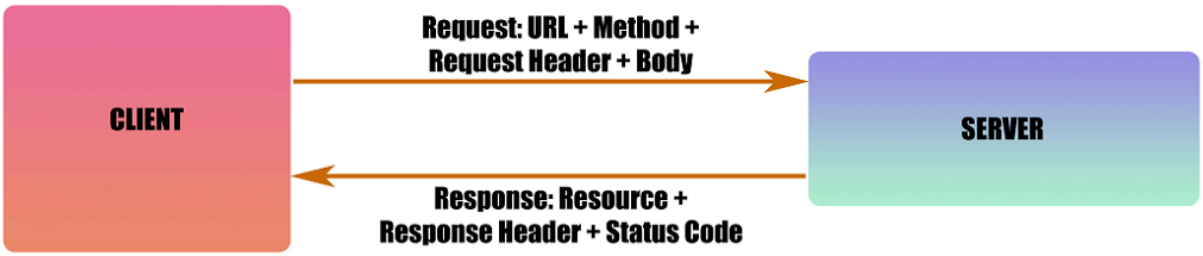
Eg: postman-api-learner.glitch.me/info?id=3422

Generally nothing is sent with the body.

So in a nutshell here is what each of these request types maps to:

GET	Read or retrieve data
POST	Add new data
PUT	Update data that already exists
DELETE	Remove data

Resources, Parameters, and Headers:



Resources:

The fundamental concept in any RESTful API is the *resource*. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it. It represents the collection, which can be accessed from the server.

Examples:
google.com/maps
google.com/search

Parameters:

Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response. There are several types of parameters: header parameters, path parameters, and query string parameters. Like a sub-resource.

Examples:
https://google.com/docs/e818931

https://www.google.com/search?ei=FNwQYLCVKMnWz7sPjMqxwA0&q=billennium+it+services&oq=billennium+IT+&gs_lcp=CgZwc3ktYWIQARgAMgIIADoECAAQRzoICC4QxwEQrwE6BAgAEEM6CwguEMcBEK8BEJMCOgYIABAWEB5Q4ldY22JgqXFoAHACeACAAawBiAH3BJIBAzAuNJgBAKABAaoBB2d3cy13aXrIAQjAAQE&sclient=psy-ab

Request Headers

Headers in the request contain meta-information about the request. It allows the client to inform the server what format of the resource it is accepting, what encoding technique for the resource it is accepting, what language it is accepting, and many more tiny details that the server might need while creating and sending the response.

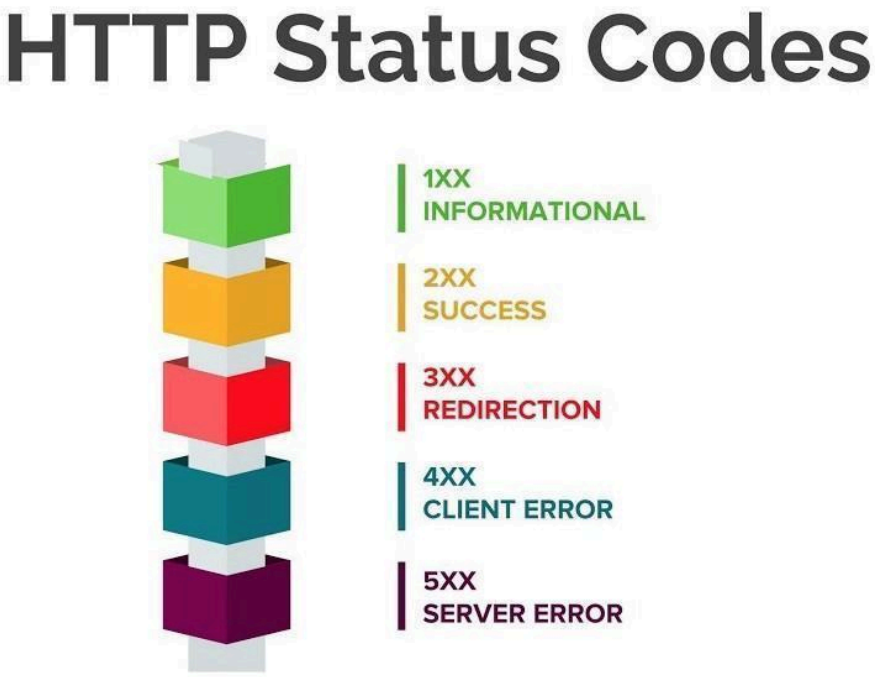
- **Accept:** This header is used to propose what content types the client would understand. The server negotiates with the client with one of these content types by sending the Content-type option in the response headers. The default one used is */* for accepting any type. Other values used are application/XML or application/json.

- **Accept-Encoding:** This header is used to propose what encoding technique for the content client understands. The server negotiates with the Content-Encoding option in one of the response headers. Values that can be used are gzip, compress, br, etc.

- **Authorization:** This contains the credentials used to authenticate the client with the server.

There are many more header fields available. And different headers are available for Response as well.

Response codes:



REST APIs use the Status-Line part of an HTTP response message to inform clients of their request's overarching result.
The status codes are divided into the five categories.

- **1xx: Informational** – Communicates transfer protocol-level information.
- **2xx: Success** – Indicates that the client's request was accepted successfully.
- **3xx: Redirection** – Indicates that the client must take some additional action in order to complete their request.
- **4xx: Client Error** – This category of error status codes points the finger at clients.
- **5xx: Server Error** – The server takes responsibility for these error status codes.

REST Specific Status Codes and the frequently used:

200 (OK)

It indicates that the REST API successfully carried out whatever action the client requested and that no more specific code in the 2xx series is appropriate. Unlike the 204 status code, a 200 response should include a response body. The information returned with the response is dependent on the method used in the request, for example:

- GET an entity corresponding to the requested resource is sent in the response;
- HEAD the entity-header fields corresponding to the requested resource is sent in the response without any message-body;
- POST an entity describing or containing the result of the action;
- TRACE an entity containing the request message as received by the end server.

201 (Created)

A REST API responds with the 201 status code whenever a resource is created inside a collection. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field.

202 (Accepted)

A 202 response is typically used for actions that take a long while to process. It indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, or even maybe disallowed when processing occurs.

Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persists until the process is completed.

204 (No Content)

The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation. The response MAY include new or updated metainformation in the form of entity-headers, which if present SHOULD be associated with the requested variant.

400 (Bad Request)

400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, deceptive request routing, etc.

401 (Unauthorized)

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

403 (Forbidden)

A 403 error response indicates that the client's request is formed correctly, but the REST API refuses to honor it, i.e., the user does not have the necessary permissions for the resource. A 403 response is not a case of insufficient client credentials; that would be 401 ("Unauthorized").
Authentication will not help, and the request SHOULD NOT be repeated. Unlike a 401 Unauthorized response, authenticating will make no difference.

404 (Not Found)

The 404 error status code indicates that the REST API can't map the client's URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

405 (Method Not Allowed)

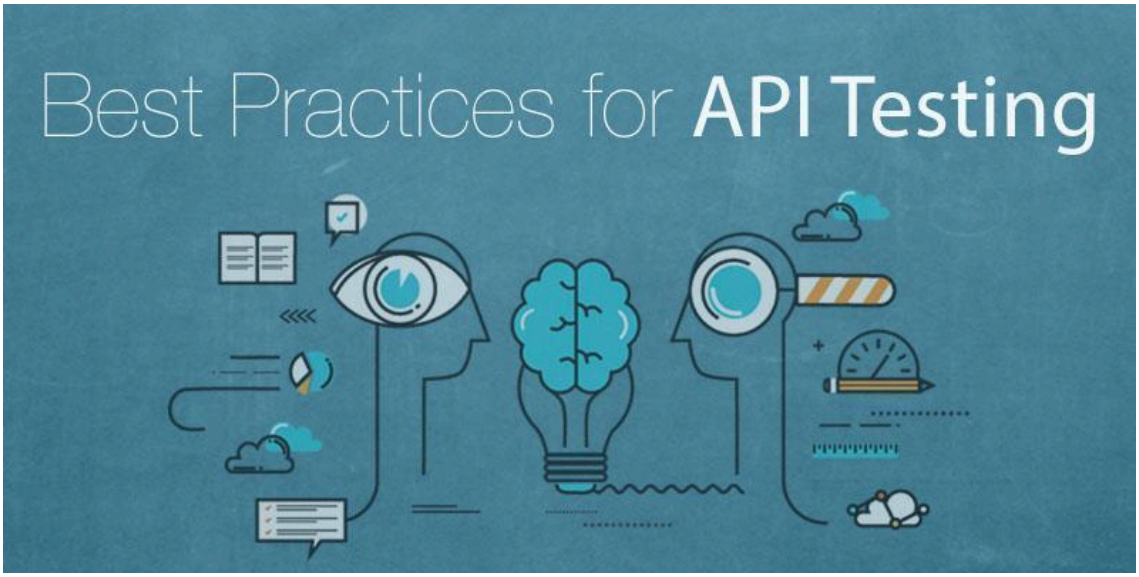
The API responds with a 405 error to indicate that the client tried to use an HTTP method that the resource does not allow. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.
A 405 response must include the Allow header, which lists the HTTP methods that the resource supports. For example: `Allow: GET, POST`

500 (Internal Server Error)

500 is the generic REST API error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.
A 500 error is never the client's fault, and therefore, it is reasonable for the client to retry the same request that triggered this response and hope to get a different response.
Please refer to https://www.restapitutorial.com/httpstatuscodes.html for all response codes.

Launch https://reqres.in/ to see some sample dummy APIs hosted.

Best Practices/Random Advices:



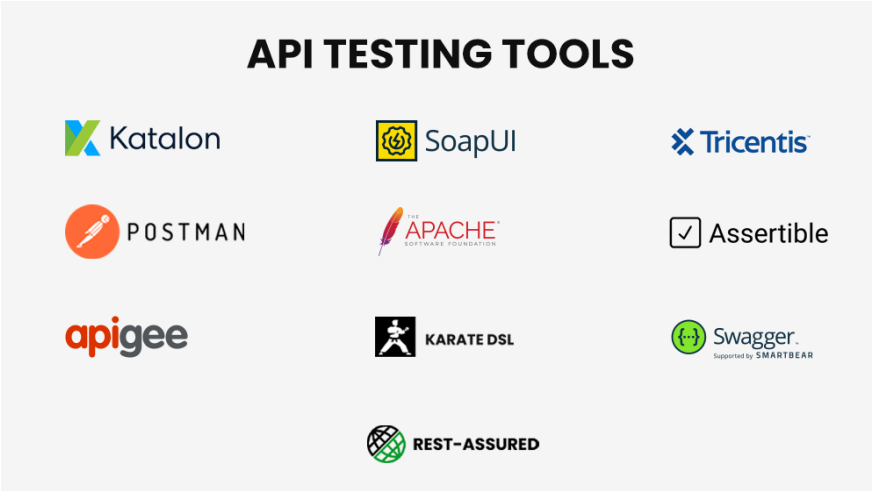
- Parameters to be used during testing should be called out in the test case
- Be careful if you're using the Delete or Purge functions – those are one-time calls
- If you're testing multiple APIs, consider the call sequencing
- Start with outlining all test scenarios before beginning; group the tests accordingly
- Try to automate the repetitive actions, which in turn increase the efficiency of testing

Challenges in API Testing:

- Initial setup for Testing

- Documentation should be standard for any organization using APIs. However, that isn't always the case. Without proper documentation, you can't adequately test
- Being able to understand and write scenarios for all of the parameter combinations and call sequencing is tough. Especially if you get a large API with many parameter options intertwined with other APIs. In this case, you can use something like Equivalence Partitioning Testing to avoid over-testing and a Decision Table to ensure hitting all the combinations
- Learning an API testing tool can be challenging
- Exception and error handling are both challenging, as those things are rarely defined for you. Need to work with your dev team to understand those exceptions
- It's intimidating. API testing is technical, some people tend to hesitate to start something new. Start with the basic testing. Use parameterization using simple GET calls

Tools available in the Market for API Testing:



Some of the tools used for API Testing are as follows:

- Postman
- Katalon Studio
- SoapUI
- Assertible
- Tricentis Tosca
- Apigee
- JMeter
- Rest-Assured
- Karate DSL

- API Fortress
- Parasoft
- HP QTP(UFT)
- vREST
- Airborne
- API Science
- APIary Inspector
- Citrus Framework
- Hippie-Swagger
- HttpMaster Express
- Mockbin
- Ping API
- Pyresttest
- Rest Console
- RoboHydra Server
- SOAPSonar
- Unirest
- WebInject

Reference Links:

https://www.katalon.com/api-testing/
https://apifriends.com/api-creation/different-types-apis/
https://itnext.io/api-calls-and-http-status-codes-e0240f78f585
https://rapidapi.com/blog/types-of-apis
https://www.mulesoft.com/resources/api/types-of-apis
https://www.programmableweb.com/news/private-partner-or-public-which-api-strategy-best-business/2014/02/21
https://techolution.com/types-of-apis/
https://www.gizmobolt.com/what-is-api-and-what-is-it-used-for-when-you-work-with-data/
https://www.omnisci.com/technical-glossary/client-server