API TESTING

TESTING AT BUSINESS LAYER, instead of UI & Database

"Testing of Application Programming Interface, which specifies how one component should interact with the other, i.e., testing of request-response mechanism."

Introduction

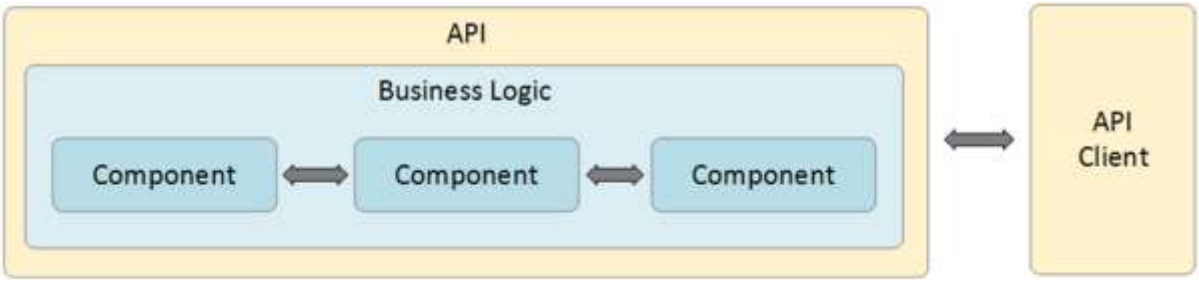
What is an API?

API is an acronym for Application Programming Interface. API (Application Programming Interface) is a computing interface which enables communication and data exchange between two separate software systems. In software application development, API is the middle layer between the presentation (UI) and the database layer. APIs enable communication and data exchange from one software system to another.

The core functionality of the system is contained within the "business logic" layer as a series of discrete but connected business components. They are responsible for taking information from the various user interfaces (UIs), performing calculations and transactions on the database layer and then presenting the results back to the user interface.

However, in addition to communicating with human beings via the UI layer, computers systems have to be able to communicate directly with each other. For example, your mobile ride sharing application will need to communicate with the mapping service, the traffic and weather services and other specialized applications used by the drivers providing the rides. In the modern, interconnected world, we take for granted that all these different systems can speak to each other seamlessly, in reality that would not be possible without APIs.

API defines requests that can be made, how to make requests, data formats that can be used, etc. between two software systems.



Why APIs? The importance of APIs is that it lets different organizations create software applications that rely on other application and services without having to constantly update their application when the internals of the dependent applications or services change. As long as the API itself remains stable, the internal implementation can change. This is an important feature of APIs, they consist of a part that doesn't change – "the interface or contract" that specifies the operations, data formats and behaviours and the implementation that can change as needed.

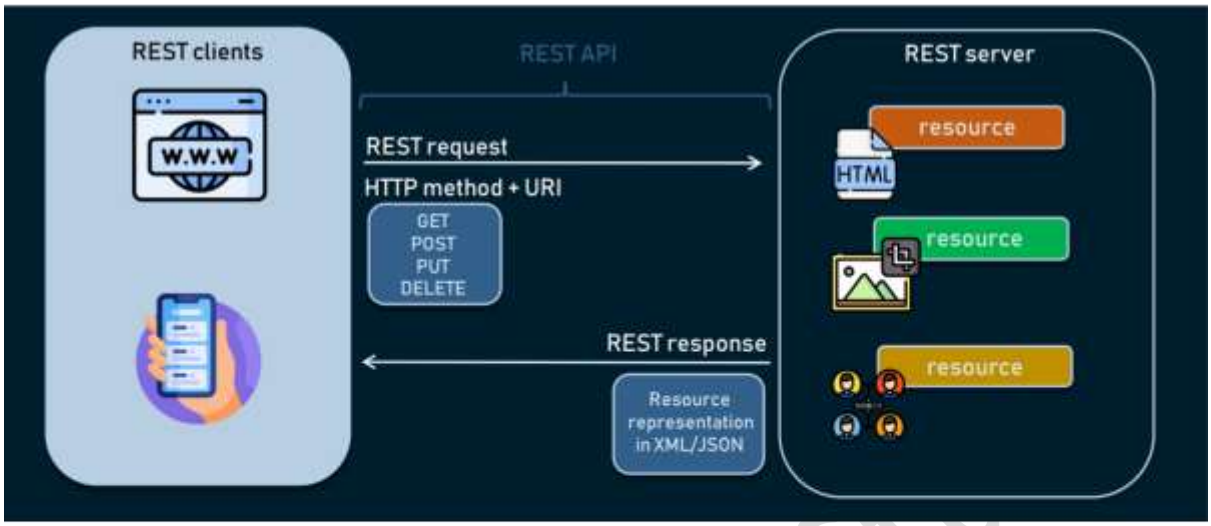
Kind of APIs

Over the years, APIs have evolved from simple code libraries that applications could use to run code on the same computer, to remote APIs that can be used to allow code on one computer to call code hosted somewhere else. Here is a quick list of the more common API technologies that exist in approximate chronological order:

- TCP/IP Sockets
- Remote Procedure Call (RPC)
- Common Object Request Broker Architecture (CORBA)
- Java Remote Method Invocation (RMI) and Enterprise Java Beans (EJBs)
- Microsoft Distributed Component Object Model (DCOM) – also known as ActiveX
- Web Services (SOAP then REST)

When looking at an API testing tool, it is important to understand which API technologies you will be using and how best to test them. Nowadays most APIs you will come across will be of the Web Service variety (either REST or SOAP), but you may come across other technologies such as Java EJBs or Microsoft DCOM/ActiveX DLLs.

What is Resource?



Resource is a physical or virtual component, which supports/extends/represents/constructs a system. Example - Wheels, breaks, headlight, fuel tank etc all are resources of a Motor bike. All resources have names, location and identifiers. Take another example of Google.

- Append "/maps" in last of base URL "https://www.google.com/" as "https://www.google.com/maps". Google will launch Google Maps.
- Try same with "/news". Google will launch Google News.

Here "Maps" and "News" are actually resources which has its names and locator.

Web Services

A Web service is a unit of managed code that can be remotely invoked using HTTP, that is, it can be activated using HTTP requests. So, Web Services allows you to expose the functionality of your existing code over the network. Once it is exposed on the network, other application can use the functionality of your program.

There are two broad classes of web service:

- Simple Object Access Protocol (SOAP)
- REpresentational State Transfer (REST)

API vs. Web Services

Yeah, API and Web Services are sometimes confusing. But always remember,

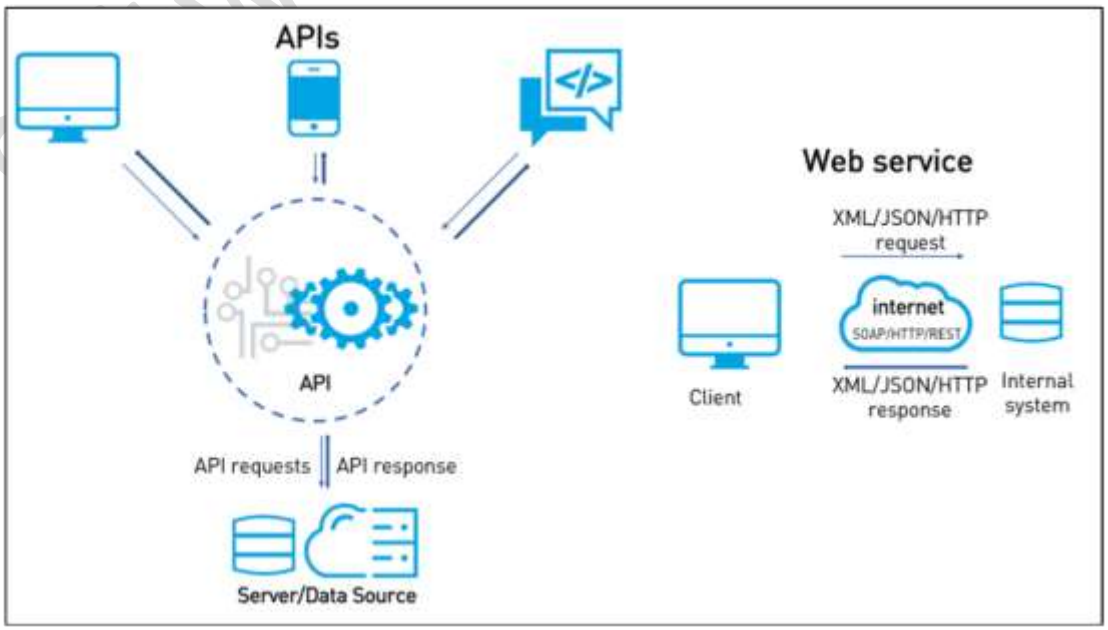
"All Web Services are APIs, but not all APIs are Web services"

How?

- API: Application interface, meaning that one application is able to interact with another application in a standard way – be it over the network or locally.
- Web Service: a type of API, which MUST BE accessed through a network connection.

Spot the difference? Yeah, the network!

APIs can be exposed through local files (such as a JAR file in a Java program, .H file in C/C++ programs, etc.) to allow two local applications to communicate with each other. This doesn't require a network as the two applications are communicating within a single device.



Note: REST APIs are a standardized architecture for building web APIs using HTTP methods, i.e., they are a type of Web Service since they use HTTP network.

SOAP Web Services

SOAP web services make use of the Web Service Definition Language (WSDL) and communicate using HTTP POST requests. They are essentially a serialization of RPC object calls into XML that can then be passed to the web service. The XML passed to the SOAP web services needs to match the format specified in the WSDL.

REST Web Services

A RESTful web API (also called a RESTful web service) is a web API implemented using HTTP and REST principles. Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs. This is because REST is an architectural style, unlike SOAP, which is a protocol.

Typically REST web services expose their operations as a series of unique "resources" which correspond to a specific URL. Each of the standard HTTP methods (POST, GET, PUT and DELETE) then maps into the four basic CRUD (Create, Read, Update and Delete) operations on each resource.

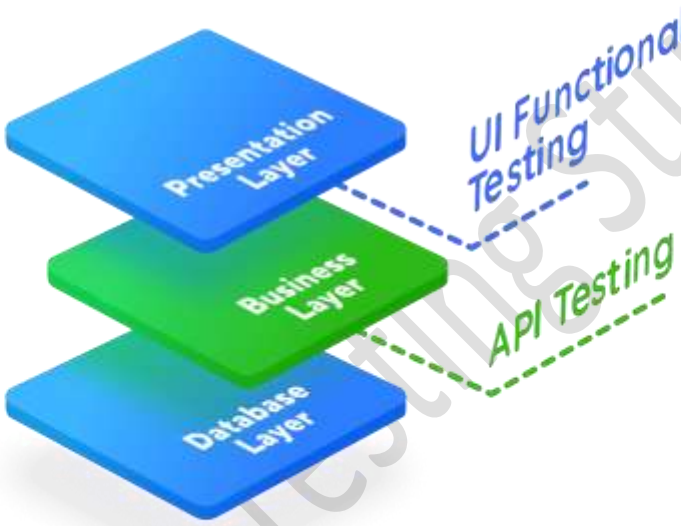
REST web services can use different data serialization methods (XML, JSON, RSS, etc.). A common format used in web browser-based APIs is JSON since it returns the data as JavaScript Object Notation (JSON) object which can be used directly in a web browser since it matches the format used by JavaScript to store arrays and objects. It is also a very compact format, making it ideal for communications on mobile networks with limited bandwidth.

#	SOAP	REST
1	A XML-based message protocol	An architectural style protocol
2	Uses WSDL for communication between consumer and provider	Uses XML or JSON to send and receive data
3	Invokes services by calling RPC method	Simply calls services via URL path
4	Does not return human readable result	Result is readable which is just plain XML or JSON
5	Transfer is over HTTP. Also uses other protocols such as SMTP, FTP, etc.	Transfer is over HTTP only
6	JavaScript can call SOAP, but it is difficult to implement	Easy to call from JavaScript
7	Performance is not great compared to REST	Performance is much better compared to SOAP - less CPU intensive, leaner code etc.

API Testing

What is API Testing?

API testing is a practice that tests the APIs directly — from their functionality, reliability, performance, to security. There are three separate layers in a typical app: the presentation (or user interface) layer, the business layer, and the database layer for Modeling and manipulating data. API testing is performed at the most critical layer: business, in which business logic processing is carried out and all transactions between the user interface and database layers happen.



While there are many aspects of API testing, it generally consists of making requests to a single or sometimes multiple API endpoints and validate the response - whether for performance, security, functional correctness, or just a status check.

API testing is used to determine whether APIs return the correct response (in the expected format) for a broad range of feasible requests, react properly to edge cases such as failures and unexpected/extreme inputs, deliver responses in an acceptable amount of time, and respond securely to potential security attacks.

API testing flow is quite simple with three main steps:

- Send the request with necessary input data.
- Get the response having output data.
- Verify that the response returned as expected in the requirement.

API vs UI Testing

API Testing is different than other software testing types as GUI is not available, and yet you are required to setup initial environment that invokes API with a required set of parameters and then finally examines the test result.

API Testing is recognised as being more suitable for test automation and continuous testing (especially the automation used with Agile software development and DevOps) than GUI testing,

- **System complexity**: GUI tests can't sufficiently verify functional paths and back-end APIs/services associated with multitier architectures. APIs are considered the most stable interface to the system under test.
- **Short release cycles with fast feedback loops**: Agile and DevOps teams working with short iterations and fast feedback loops find that GUI tests require considerable rework to keep pace with frequent change. Tests at the API layer are less brittle and easier to maintain.

For these reasons, it is recommended that teams increase their level of API testing while decreasing their reliance on GUI testing.

Benefits of API Testing

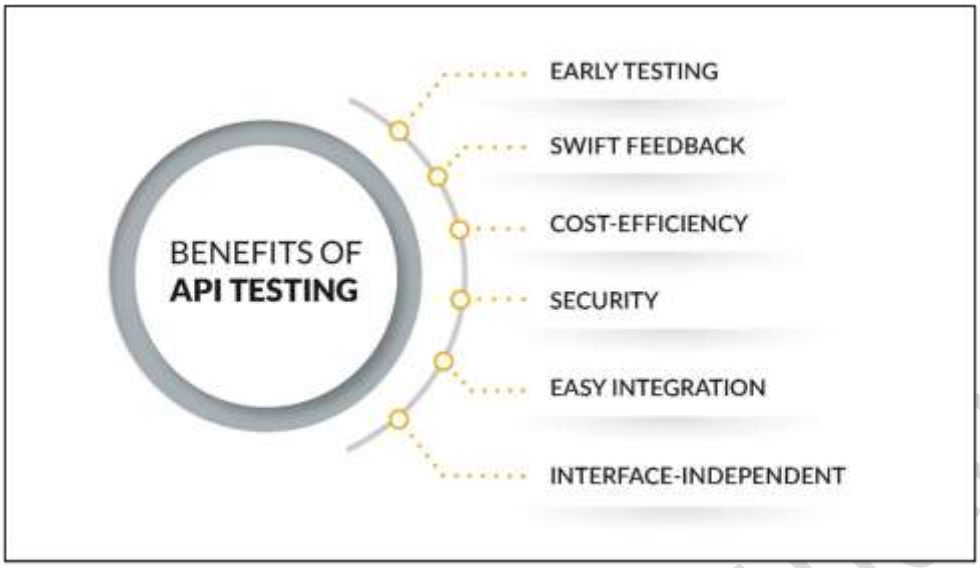
Language-independent: Data is exchanged via XML and JSON formats, so any language can be used for test automation. XML and JSON are typically structured data, making the verification fast and stable. There are also built-in libraries to support comparing data using these data formats.

GUI-independent: API testing can be performed in the app prior to GUI testing. Early testing means early feedback and better team productivity.

Improved test coverage: Most API/web services have specifications, allowing you to create automated tests with high coverage — including functional testing and non-functional testing.

Faster releases: It is common that executing API testing saves up to eight hours compared to UI testing, allowing software development teams to release products faster.

Earlier Testing: With API testing, once the logic is designed, tests can be built to validate the correctness in responses and data. We don't have to wait for various teams to finish their work or for full applications to be built.



Easier Test Maintenance: UIs are constantly changing and moving around based on how they are accessed - browsers, devices, screen orientation, etc. This creates a nightmare scenario where tests are being constantly rewritten to keep up with the actual code in production. API changes are much more controlled and infrequent - often times API definition files like OpenAPI Spec can help make refactoring tests quite easy.

Faster Time to Resolution: When API tests fail, we know exactly where our system broke and where the defect can be found. This helps reduce time triaging bugs between builds, integrations, and even different team-members.

Speed and Coverage: 300 UI tests may take 30 hours to run. 300 API tests could be run in 3 minutes. That means you'll find more bugs in less time, while also being about to fix them immediately. API test automation requires less code than automated GUI tests, resulting in faster testing and a lower overall cost.

API tests can be integrated with GUI tests. For example, integration can enable new users to be created within the app before a GUI test is performed.

API Testing Types

Functional testing: Includes testing particular functions in the codebase. These features are the representation of specific scenarios to make sure the API functions are handled well within the planned parameters.

UI testing: UI testing is defined as a test of the user interface for the API and other integral parts. UI testing focuses more on the interface which ties into the API rather than the API testing itself. Although UI testing is not a specific test of API in terms of codebase, this technique still provides an overview of the health, usability, and efficiency of the app's front and back ends.

Security testing: This practice ensures the API implementation is secure from external threats. Security testing also includes additional steps such as validation of encryption methodologies, and of the design of the API access control. It also includes user rights management and authorization validation.

Load testing: Load testing generally occurs after a specific unit or the whole codebase has been completed. This technique checks if the theoretical solutions work as planned. Load testing monitors the app's performance at both normal and peak conditions.

Runtime and error detection: This testing type is related to the actual running of the API — particularly with the universal results of utilizing the API codebase. This technique focuses on one of the below aspects: monitoring, execution errors, resource leaks, or error detection.

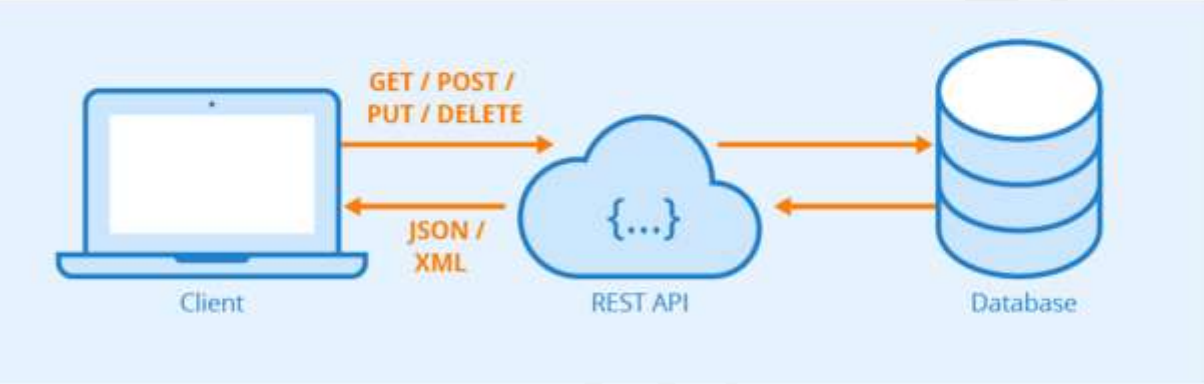
Penetration testing: Penetration testing is considered the second test in the auditing process. In this type, users with limited API knowledge will try to assess the threat vector from an outside perspective, which is about functions, resources, processes, or aim to the entire API and its components.

Fuzz testing: Fuzz testing is another step in the security audit process. In fuzz testing, a vast amount of random data (referred to as "noise" or "fuzz") will be input into the system to detect any forced crashes or negative behaviours. This technique tests the API's limits to prepare for the "worst-case scenarios".

HTTP Request

HTTP Request

The internet boasts a vast array of resources hosted on different servers. For you to access these resources, your browser needs to be able to send a request to the servers and display the resources for you. HTTP (Hypertext Transfer Protocol), is the underlying format that is used to structure request and responses for effective communication between a client and a server. The message that is sent by a client to a server is what is known as an HTTP request. When these requests are being sent, clients can use various methods.



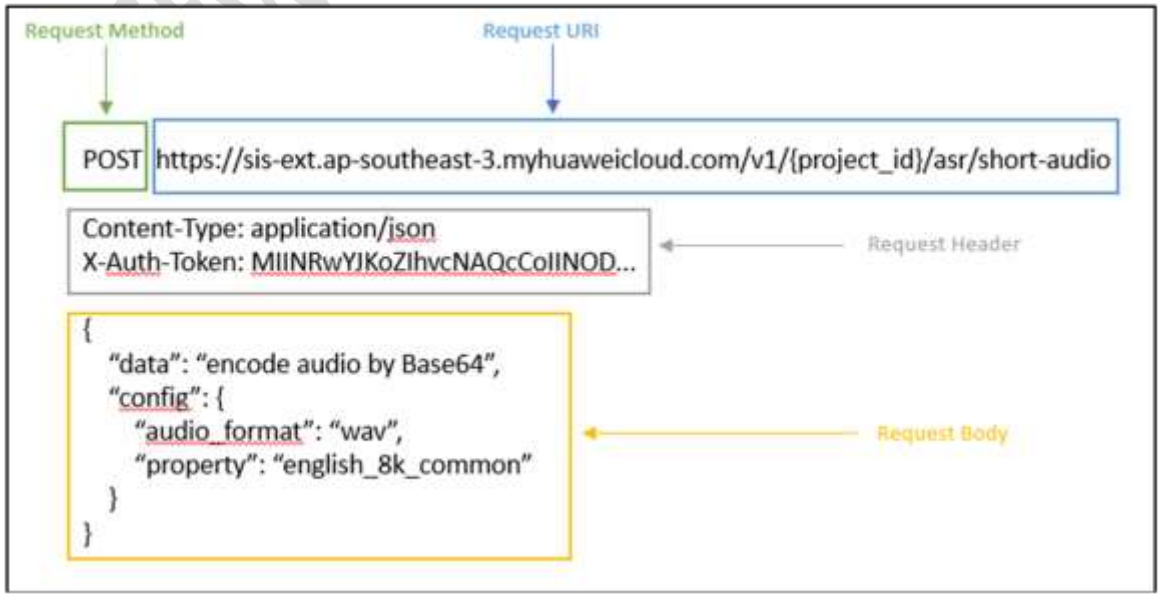
An HTTP request method is an action to be performed on a resource identified by a given Request-URL. Request methods are case-sensitive, and should always be noted in upper case. The client submits an HTTP request to the server, and after internalizing the message, the server sends back a response. The response contains status information about the request.

JSON

JSON (JavaScript Object Notation) is a common format for sending and requesting data through a REST API. A JSON object looks like a JavaScript Object. In JSON, each property and value must be wrapped with double quotation marks, like this:

Components of an HTTP request

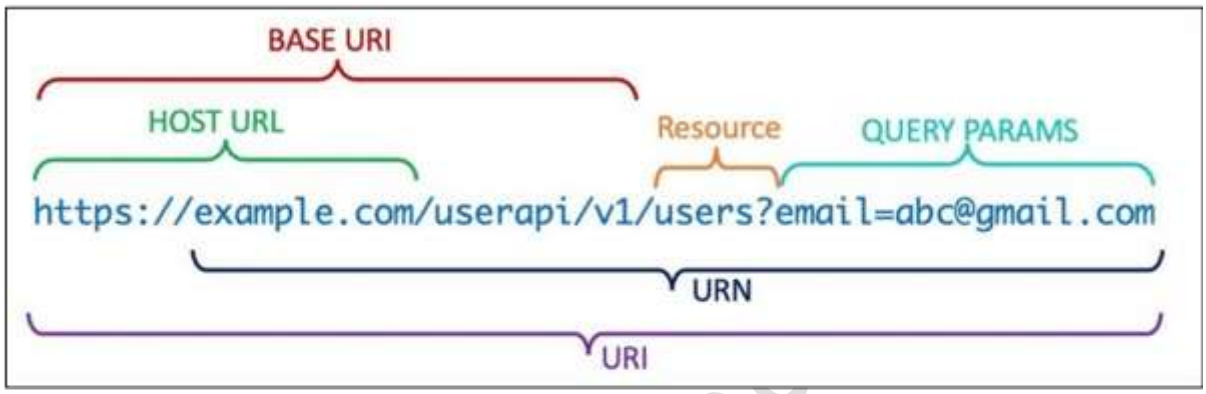
- **Action** showing HTTP method like GET, PUT, POST, DELETE.
- Uniform Resource Identifier (**URI**): URI is the identifier for the resource on the server.
- **HTTP version**: Indicate the HTTP version like- HTTP V1.1.
- **Request Header**: Request Header carries metadata for the HTTP request message. Metadata could be a client type, format supported by the client, format of a message body, cache setting etc.
- **Request Body**: Resource body indicates message content or resource representation.



Request Endpoint/URI

It's important to know that a request is made up of four things: The endpoint, The method, The headers and the data (or body). The endpoint (or route) is the URL you request for.

The **ROOT-ENDPOINT** is the starting point of the API you're requesting from. The root-endpoint of GitHub API is https://api.github.com while the root-endpoint Twitter's API is https://api.twitter.com



The PATH determines the resource you're requesting for. Think of it like an automatic answering machine that asks you to press 1 for a service, press 2 for another service, 3 for yet another service and so on. You can access paths just like you can link to parts of a website.

For example, to get a list of all posts tagged under "API Testing" on Stack Overflow, you navigate to https://www.stackoverflow.com/tag/apitesting/.
https://www.stackoverflow.com/ is the root-endpoint and /tag/apitesting is the path.

To understand what paths are available to you, you need to look through the API documentation.

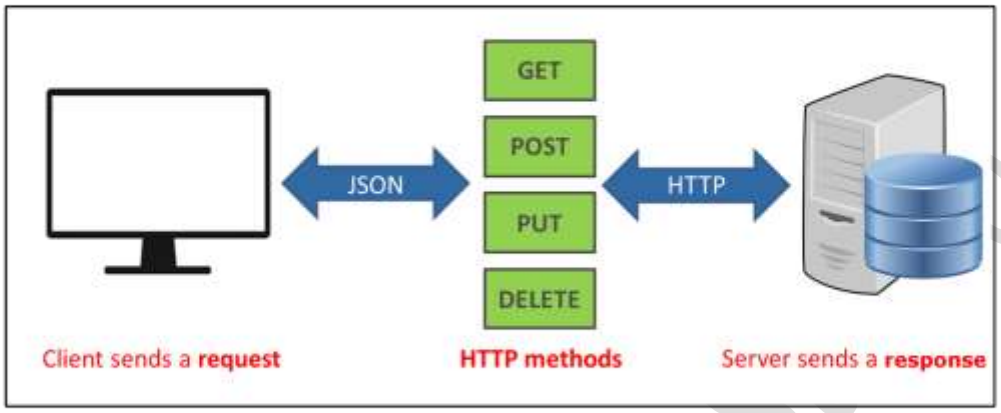
The final part of an endpoint is **QUERY PARAMETERS**. Query parameters give you the option to modify your request with key-value pairs. They always begin with a question mark (?). Each parameter pair is then separated with an ampersand (&), like this:

?query1=value1&query2=value2

When you try to get a list of a user's repositories on GitHub - https://api.github.com/users/sts/repos?sort=pushed

HTTP Request Methods

Having a basic understanding of the different HTTP methods, or verbs an API supports is an helpful knowledge when exploring and testing APIs.



GET

The most common and widely used method in APIs and websites. Simply put, the GET method is used to retrieve data from a server at the specified resource. For example, say you have an API with a /users endpoint. Making a GET request to that endpoint should return a list of all available users.

Since a GET request is only requesting data and not modifying any resources, it's considered a safe and idempotent method. Example URIs,

- *HTTP GET http://www.appdomain.com/users*
- *HTTP GET http://www.appdomain.com/users?size=20&page=5*
- *HTTP GET http://www.appdomain.com/users/123*
- *HTTP GET http://www.appdomain.com/users/123/address*

When you're creating tests for an API, the GET method will likely be the most frequent type of request made by consumers of the service, so it's important to check every known endpoint with a GET request.

- Check that a valid GET request returns a 200 status code.
- Ensure that a GET request to a specific resource returns the correct data. For example, GET /users returns a list of users.
- GET is often the default method in HTTP clients, so creating tests for these resources should be simple with any tool you choose.

POST

Used to send data to the API server to create or update a resource. The data sent to the server is stored in the 'request body' of the HTTP request.

The simplest example is a contact form on a website. When you fill out the inputs in a form and hit send, that data is put in the body of the request and sent to the server. This may be JSON, XML, or query parameters (there's plenty of other formats, but these are the most common).

It's worth noting that a POST request is non-idempotent. It mutates data on the backend server (by creating or updating a resource), as opposed to a GET request which does not change any data. Example URIs,

- *HTTP POST http://www.appdomain.com/users*
- *HTTP POST http://www.appdomain.com/users/123/accounts*

Since POST requests modify data, it's important to have API tests for all of your POST methods.

- Create a resource with a POST request and ensure a 200-status code is returned.
- Next, make a GET request for that resource, and ensure the data was saved correctly.
- Add tests that ensure POST requests fail with incorrect or ill-formatted data.

PUT

Similar to POST, PUT requests are used to send data to the API to update or create a resource. The difference is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly might have side effects of creating the same resource multiple times. Generally, when a PUT request creates a resource, the server will respond with a 201 (Created), and if the request modifies existing resource the server will return a 200 (OK) or 204 (No Content). Example URIs,

- *HTTP PUT http://www.appdomain.com/users/123*
- *HTTP PUT http://www.appdomain.com/users/123/accounts/456*

Testing an APIs PUT methods is very similar to testing POST requests. But now that we know the difference between the two (idempotency), we can create API tests to confirm this behaviour.

- Repeatedly calling a PUT request always returns the same result (idempotent).
- The proper status code is returned when creating and updating a resource (e.g., 201 or 200/204).
- After updating a resource with a PUT request, a GET request for that resource should return the correct data.
- PUT requests should fail if invalid data is supplied in the request -- nothing should be updated.

The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections, whereas PUT requests are made on a single resource.

PATCH

One of the lesser-known HTTP methods, it is similar to POST and PUT. The difference with PATCH is that you only apply partial modifications to the resource. The difference between PATCH and PUT, is that a PATCH request is non-idempotent (like a POST request).
To expand on partial modification, say an API has a /users/{{userid}} endpoint, and a user has a username. With a PATCH request, you may only need to send the updated username in the request body - as opposed to POST and PUT which require the full user entity.
Example,

HTTP GET /users/1 - produces below response:
{ "id": 1, "username": "admin", "email": "email@example.org"}

A sample patch request to update the email will be like this:

HTTP PATCH /users/1
[{ "op": "replace", "path": "/email", "value": "new.email@example.org" }]

- A successful PATCH request should return a 2xx status code.
- PATCH requests should fail if invalid data is supplied in the request -- nothing should be updated.
- The semantics of PATCH requests will largely depend on the specific API you're testing.

DELETE

The DELETE method is exactly as it sounds: delete the resource at the specified URL. This method is one of the more common in RESTful APIs so it's good to know how it works. If a new user is created with a POST request to /users, and it can be retrieved with a GET request to /users/{{userid}}, then making a DELETE request to /users/{{userid}} will completely remove that user. Example URIs,

- *HTTP DELETE http://www.appdomain.com/users/123*
- *HTTP DELETE http://www.appdomain.com/users/123/accounts/456*

DELETE requests should be heavily tested since they generally remove data from a database. Be careful when testing DELETE methods, make sure you're using the correct credentials and not testing with real user data.

- Create a new user with a POST request to /users
- With the user id returned from the POST, make a DELETE request to /users/{{userid}}
- A subsequent GET request to /users/{{userid}} should return a 404 not found status code.
- In addition, sending a DELETE request to an unknown resource should return a non-200 status code.

HEAD

The HEAD method is almost identical to GET, except without the response body. In other words, if GET /users returns a list of users, then HEAD /users will make the same request but won't get back the list of users. HEAD requests are useful for checking what a GET request will return before actually making a GET request -- like before downloading a large file or response body.

It's worth pointing out that not every endpoint that supports GET will support HEAD - it completely depends on the API you're testing.

Making API requests with HEAD methods is actually an effective way of simply verifying that a resource is available. It is good practice to have a test for HEAD requests everywhere you have a test for GET requests (as long as the API supports it).

- Verify and check HTTP headers returned from a HEAD request
- Make assertions against the status code of HEAD requests
- Test requests with various query parameters to ensure the API responds
- Another useful case for HEAD requests is API smoke testing - make a HEAD request against every API endpoint to ensure they're available.

OPTIONS

Not as widely used as the other HTTP methods, an OPTIONS request should return data describing what other methods and operations the server supports at the given URL.

OPTIONS requests are more loosely defined and used than the others, making them a good candidate to test for fatal API errors. If an API isn't expecting an OPTIONS request, it's good to put a test case in place that verifies failing behaviour.

- Primarily, check the response headers and status code of the request
- Test endpoints that don't support OPTIONS, and ensure they fail appropriately

Other Methods,

- CONNECT: establish a network connection to a web server.
- TRACE: performs a message loop-back test.

API Header

Headers are the name or value pairs that are displayed in the API request and response messages.

- **Request Header**: Contains information such as the type, capabilities, and version of the browser that generates the request, the operating system used by the client, the page that was requested, the various types of outputs accepted by the browser, and so on.

- **Response Header**: Includes information such as the type, date, and size of the file sent back by the server, as well as information regarding the server.

Why they are important? Headers define the operating parameters for the process of data transfer from source to destination. They help to know about the source and destination, decide the data transmission mode and encoding type. Headers also speak aloud about the content, its type, its size, restricts amount and variety of data allowed at both ends of communication.

Rest API Header Types

Headers are mostly classified as request and response headers - property-value pairs that are separated by a colon. Set the request headers when you are sending the request > Set the assertion against the response headers. Some header types,

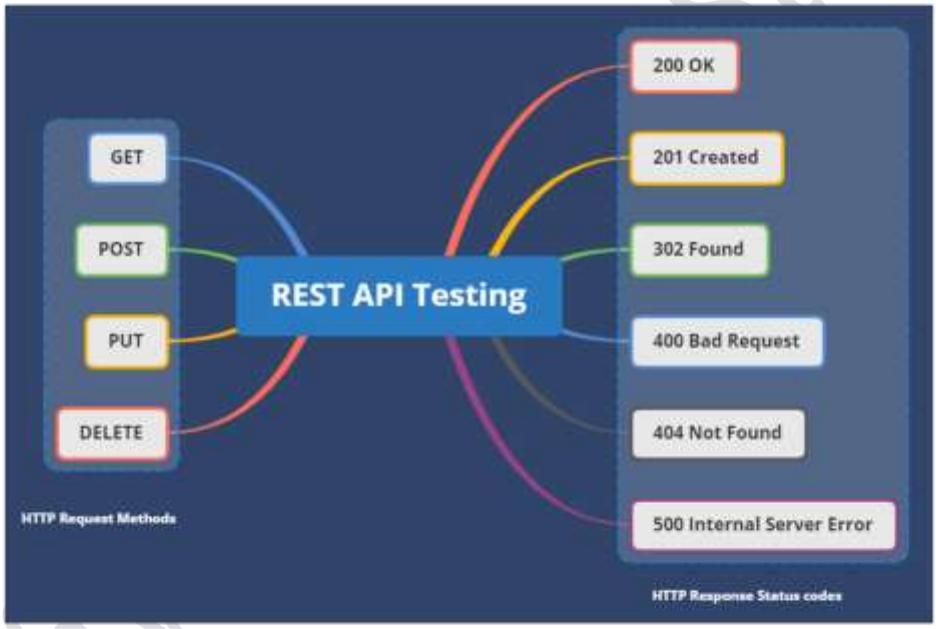
- **Authorization**: Carries credentials containing the authentication information of the client for the resource being requested.
- **Content-Type**: The MIME type of the request or response. E.g., text/html or text/JSON.
- **Date**: The date and time of the request or response.
- **Accept**: Informs the server about the types of data that can be sent back. E.g., Accept-Charset: which character sets are acceptable by the client.
- **WWW-Authenticate**: Sent by the server if it needs a form of authentication before it can respond with the actual resource being requested. Often sent along with a response code of 401, which means 'unauthorized'.
- **Cache-Control**: Cache policy defined by the server for this response, a cached response can be stored by the client and re-used till the time defined by the Cache-Control header.
- **Expires**: Response header indicating the time after which the response is considered stale.
- **Set-Cookie**: Send cookies from the server to the user-agent.

HTTP Status Codes

It takes a lot of work to provide a reliable API - which consumers can trust to provide consistent, stable, meaningful, and expected behaviour. In this line - 'HTTP Status codes' are standard responses that help identify the cause of the problem when the resource request is not processed properly. HTTP response status codes are grouped in five classes:

- Informational responses (100–199),
- Successful responses (200–299),
- Redirects (300–399),
- Client errors (400–499),
- Server errors (500–599).

Some common status codes,



- **200 Ok** | Successful requests other than creations and deletions.
- **201 Created** | Successful creation.
- **204 No Content** | Successful deletion.
- **400 Bad Request** | The path info doesn't have the right format, or a parameter or request body value doesn't have the right format, or a required parameter is missing, or values have the right format but are invalid in some way.
- **403 Forbidden** | The invoker is not authorized to invoke the operation.
- **404 Not Found** | The object referenced by the path does not exist.
- **405 Method Not Allowed** | The method is not one of those allowed for the path.
- **409 Conflict** | An attempt was made to create an object that already exists.
- **500 Internal Server Error** | The execution of the service failed in some way.

HTTP Status Codes aren't just a nice to have thing when it comes to APIs, they are essential.

API Validation

REST authentication methods

Before jumping to it, let's first understand authorization & authentication,

- **Authentication**: proving your identity. Say a company-issued I-card proves that you are an employee.
- **Authorization**: proving your right to access. With company I-card you can have access to common areas but NOT the security or server rooms. Basically, your permissions.

Some common rest authentication schemes are,

- **Basic**: The most straightforward and easiest method. Use of encoded 'username – password' in the request header.

- **Bearer**: also called 'token' authentication. Token - a cryptic string, usually generated by the server in response to a login request. Once the token is generated, client must send this token in the Authorization header when making requests to protected resources.

- **API Keys**: Generate a key [unique value assigned to each first-time user] >> then use this key the next time you try to access the system.

- **OAuth**: User signs-in, grants permission, and your app can authenticate each request with an access token. E.g., 'Login using Facebook/Gmail' option on many websites OR when you cross-post on any social network | App-A > Facebook Login popup > Enter credentials > Give permissions > App-A now has the token generated by Facebook > this token can now be used by App-A based on the permissions granted.

Things to verify in API testing

One of the common interview Q. Before that, how does an API [request-response] work OR what components are involved?

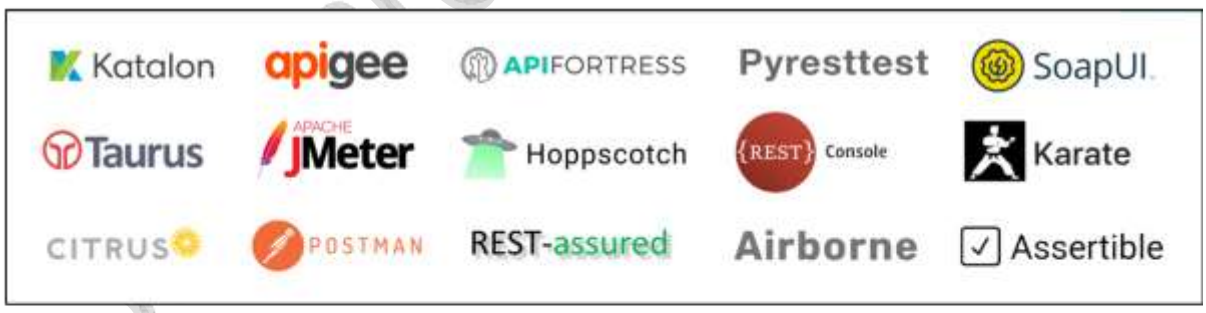
- **Endpoint**: the URI + path/query parameters to send the request to.
- **Request-type**: different HTTP methods.
- **Request**: the payload in xml/json/etc. format.
- **Auth**: check for authorization and permission.
- **Response**: response from the server.

Considering these components, what all things you would verify?

- **Schema validation**: The most important. For both request & response, i.e., different fields [type/format/occurrence/etc.] as per the specifications [XSDs]. Different permutations & combinations.
- **Test Data**: Different P&C of Test data in the API request.
- **Authentication**: verifying the auth mechanism [happy-flow/time-out/invalid/etc.]
- **HTTP methods**: which all methods are supported.
- **HTTP Status Code**: Basic check before validating the response.
- **Error conditions**: how well the API handles the invalid inputs.
- Performance & Security aspects.

API Testing Tools

A large variety of API testing tools is available, ranging from paid subscription tools to open-source offerings. Some specific examples of API testing tools include:



Conclusion

API testing is now considered critical for automating testing because APIs now serve as the primary interface to application logic and because GUI tests are difficult to maintain with the short release cycles and frequent changes commonly used with Agile software development and DevOps.