**API: Application Programming Interface**

API:

An application programming interface (API) is a set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications.

Webservice:

A Web Service is defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network"

Clearly, both are means of communications. The difference is that Web Service almost always involves communication over network and HTTP is the most commonly used protocol. Web service also uses SOAP, REST, and XML-RPC as a means of communication. While an API can use any means of communication e.g. DLL files in C/C++, Jar files/ RMI in java, Interrupts in Linux kernel API etc.So, you can say that-

1. Web Service is an API wrapped in HTTP.

2. All Web Services are API but APIs are not Web Services.

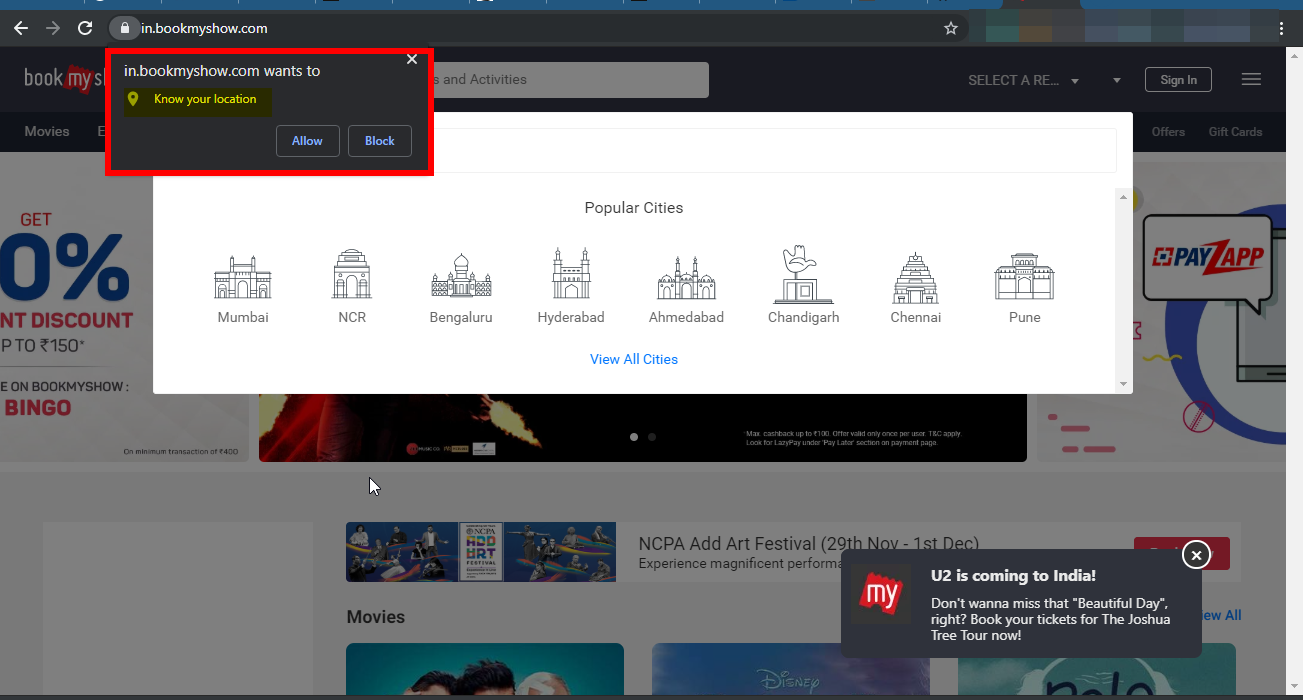
3. Web Service might not perform all the operations that an API would perform.

4. A Web Service needs a network while an API doesn't need a network for its operation.

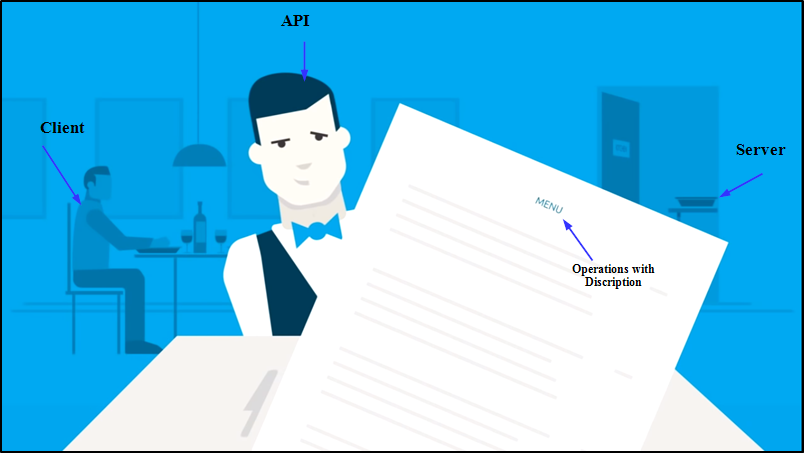
**All Web services are APIs, but all APIs are not Web services.**

API is an interface that allows your application to interact with an external service using a simple set of commands. API is like an open language, the rules of which are shared by a certain service. You can teach your application the rules of this language, so it can communicate with the service and access all the functions and data that the service is ready to share. You do not need to know the internal logic of the service, just send a simple command and the service will return the necessary data.

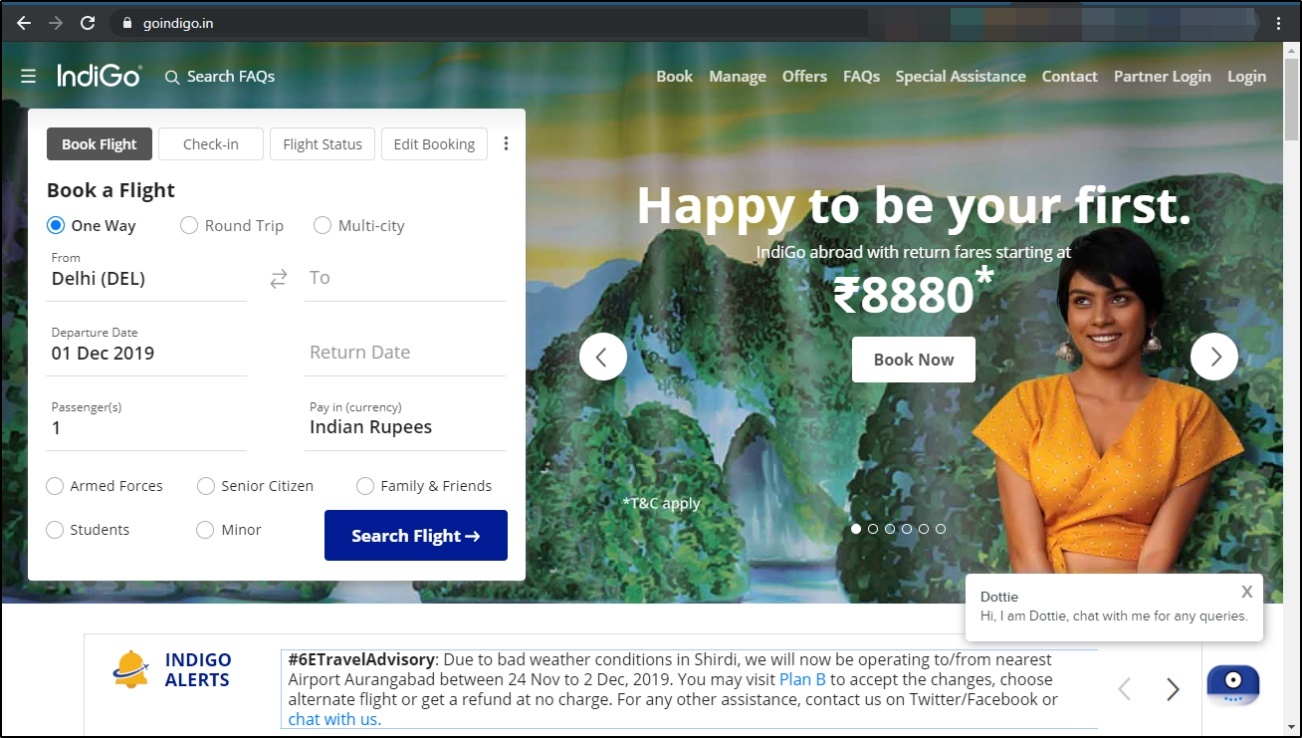
API makes the developers life easy. It lists several operations with description that developers can use. It controls resources also resources and communication between services. For example, you might have seen multiple times that your web browser wants to know your location. That’s an API call in layman terms. Figuring out your GPS location and destination is done by Application.



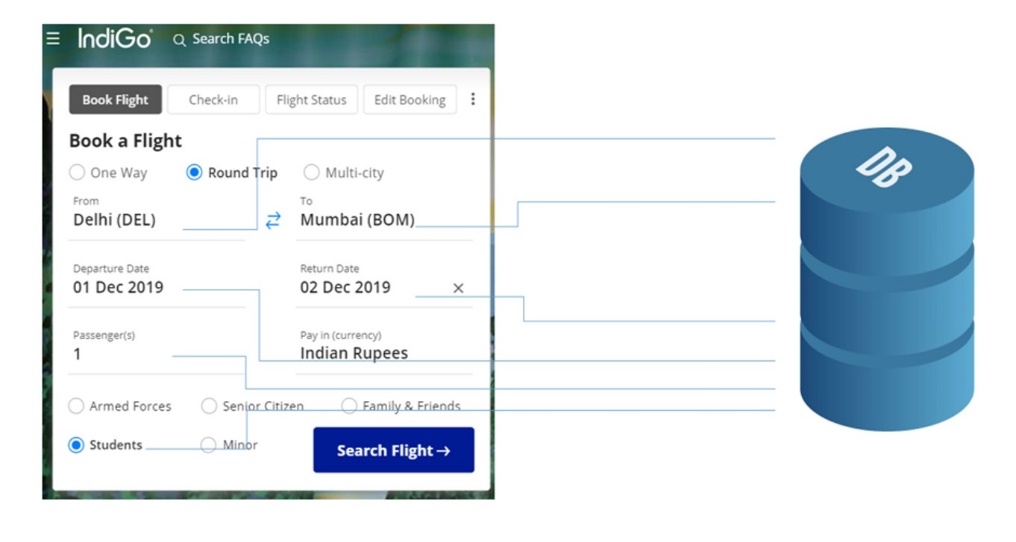
Let us take a familiar example. Imagine you’re sitting at a table in a restaurant with a menu of choices to order from. The kitchen is the part of the “system” that will prepare your order. What is missing is the critical link to communicate your order to the kitchen and deliver your food back to your table. That’s where the waiter or API comes in. The waiter is the messenger – or API – that takes your request or order and tells the kitchen – the system – what to do. Then the waiter delivers the response back to you; in this case, it is the food.



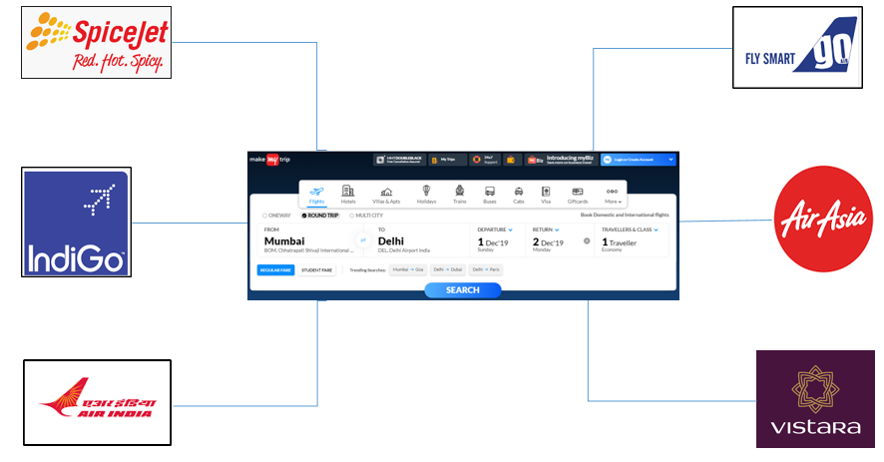
Here is a real-life API example.



You may be familiar with the process of searching flights online. Just like the restaurant, you have a variety of options to choose from, including different cities, departure and return dates, and more. Let us imagine that you’re booking you are flight on an airline website. You choose a departure city and date, a return city and date, cabin class, as well as other variables. In order to book your flight, you interact with the airline’s website to access their database and see if any seats are available on those dates and what the costs might be.



However, what if you are not using the airline’s website––a channel that has direct access to the information? What if you are using an online travel service, such as MakeMyTrip or EaseMyTrip, which aggregates information from a number of airline databases?

The travel service, in this case, interacts with the airline’s API. The API is the interface that, like your helpful waiter, can be asked by that online travel service to get information from the airline’s database to book seats, baggage options, etc. The API then takes the airline’s response to your request and delivers it right back to the online travel service, which then shows you the most updated, relevant information

.

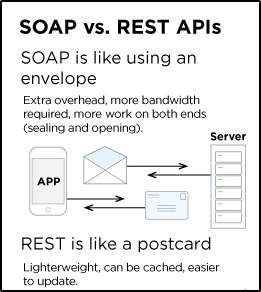
An API is different from a database backed (or static) web application or site in that it does not generally need to contain a front end — no HTML, CSS is necessary to be shown to the user via static pages or dynamically generated templates that fuse data with reusable layouts.

Requests to retrieve or write data are generally done without a front end, by sending an HTTP request to a server.

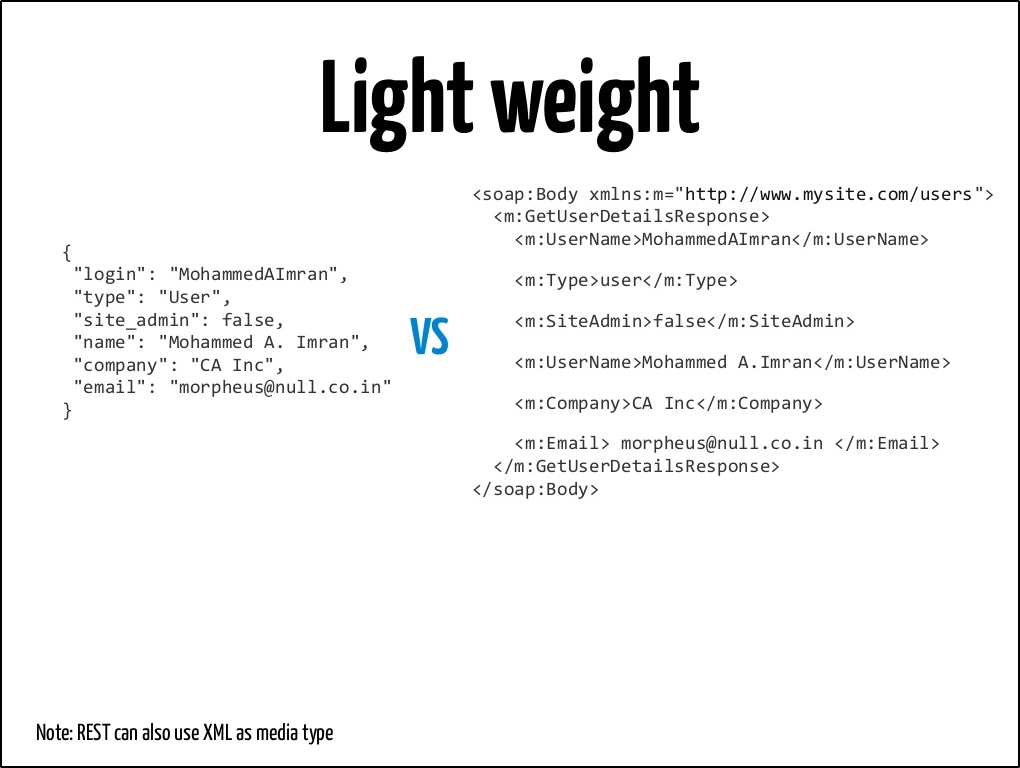
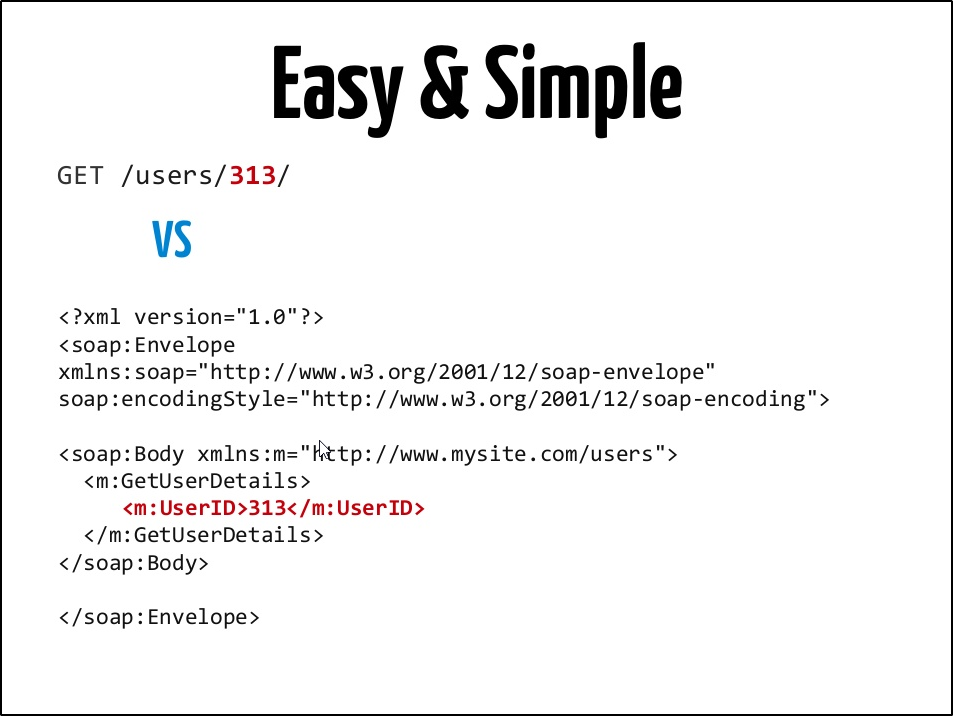
**Types of API**

**SOAP — Simple Object Access Protocol - Protocol**

**REST — Representational State Transfer – Architecture**



|  |  |  |
| --- | --- | --- |
| **Difference** | **SOAP** | **REST** |
| Style | A protocol. Having an official standard. | An architectural style. Not having an official standard. |
| Data Format | Uses: mostly HTTP and XML | Uses: HTTP, JSON, URL and XML. But the most preferred format for transferring data is JSON. |
| Javascript support | Has compatibility with JavaScript, its support with larger implementation is limited | More convenient with JavaScript |
| Bandwidth | SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.  <?xml version="1.0"?>  <SOAP-ENV:Envelope  xmlns:SOAP-ENV  ="http://www.w3.org/2001/12/soap-envelope"  SOAP-ENV:encodingStyle  =" http://www.w3.org/2001/12/soap-encoding">  <soap:Body>  <Demo.guru99WebService  xmlns="http://tempuri.org/">  <EmployeeID>int</EmployeeID>  </Demo.guru99WebService>  </soap:Body>  </SOAP-ENV:Envelope> | REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.  {"city":"Mumbai","state":"Maharastra"} |
| Security | SOAP defines its own security.  Supports WS-Security and SSL | RESTful web services inherit security measures from the underlying transport.  Supports SSL and HTTPs. |
| Data Cache | Cannot be cached | Can be cached |
| Payload Handling | Has a strict communication contract and needs knowledge of everything before any interaction | Need no knowledge of the API. |

****

**REST APIs**

**URIs**

REST APIs make use of Uniform Resource Identifiers (URIs) to access resources.

For example, <https://api.github.com/users/RocketTest>

This format is very easy to understand and is readable to a normal human being. Here, it is understandable that the client is requesting data of the user, which is **RocketTest** in this case.

**URI format**

URI = scheme "://" authority "/" path [ "?" query ] [ "#" fragment ]

<https://api.github.com/users/RocketTest/repos>

This shows repositories of RocketTest. There's a hierarchical relationship between users and their repositories.

**Request methods**

REST API and HTTP go hand in hand in aspects such as request methods, response codes, and message headers

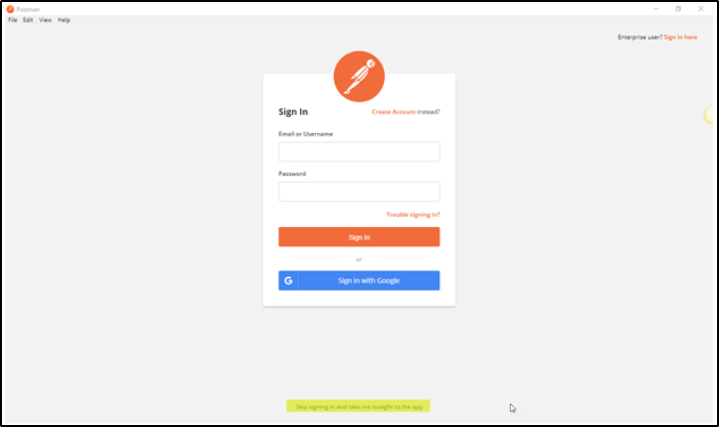
Request methods are simply HTTP methods like GET, POST, DELETE, and so on. But please note that these methods have fixed contextual meaning within REST API's resource model.

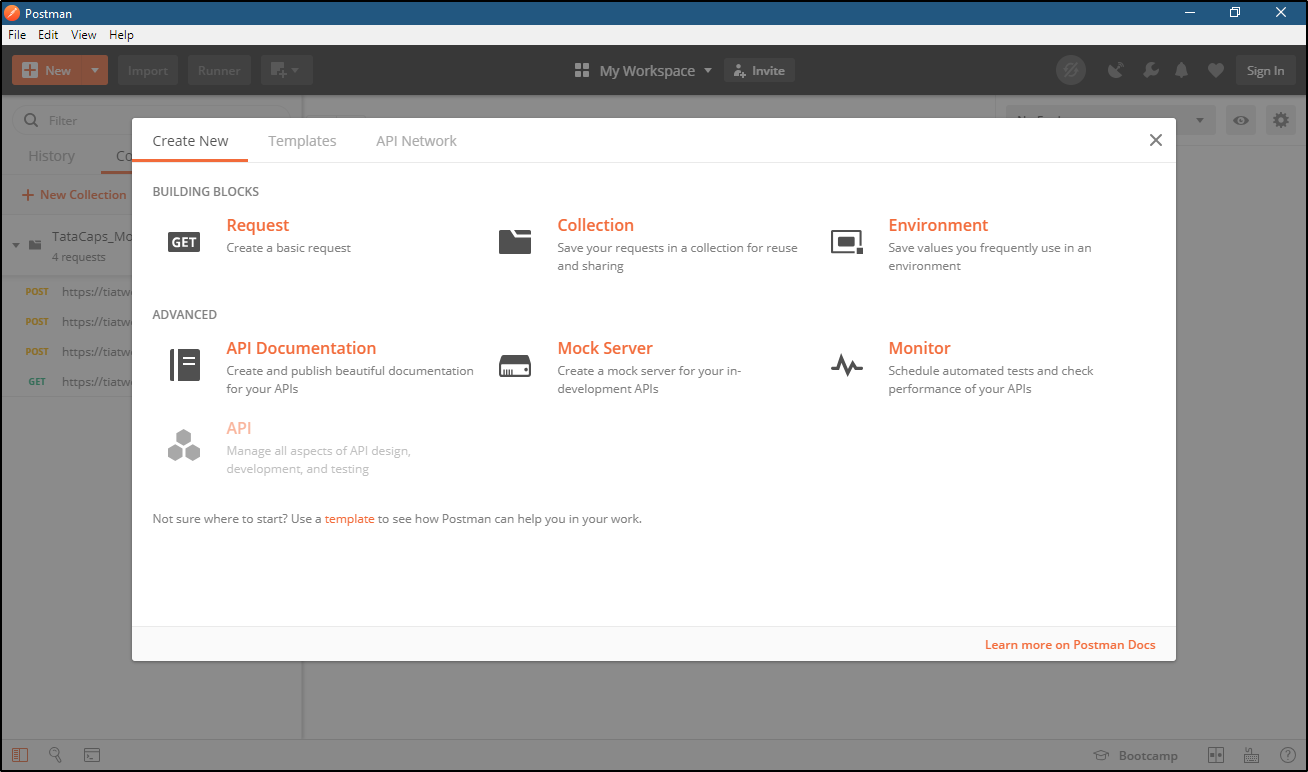
|  |  |
| --- | --- |
| **Method** | **Meaning** |
| GET | Fetches (gets) the representation of a resource's state |
| POST | Creates a new resource |
| PUT | Updates a resource |
| DELETE | Removes a resource |
| HEAD | Fetches metadata associated with a resource's state |
| OPTIONS | Lists the available methods |

**Set-Up of Tools and Environment**

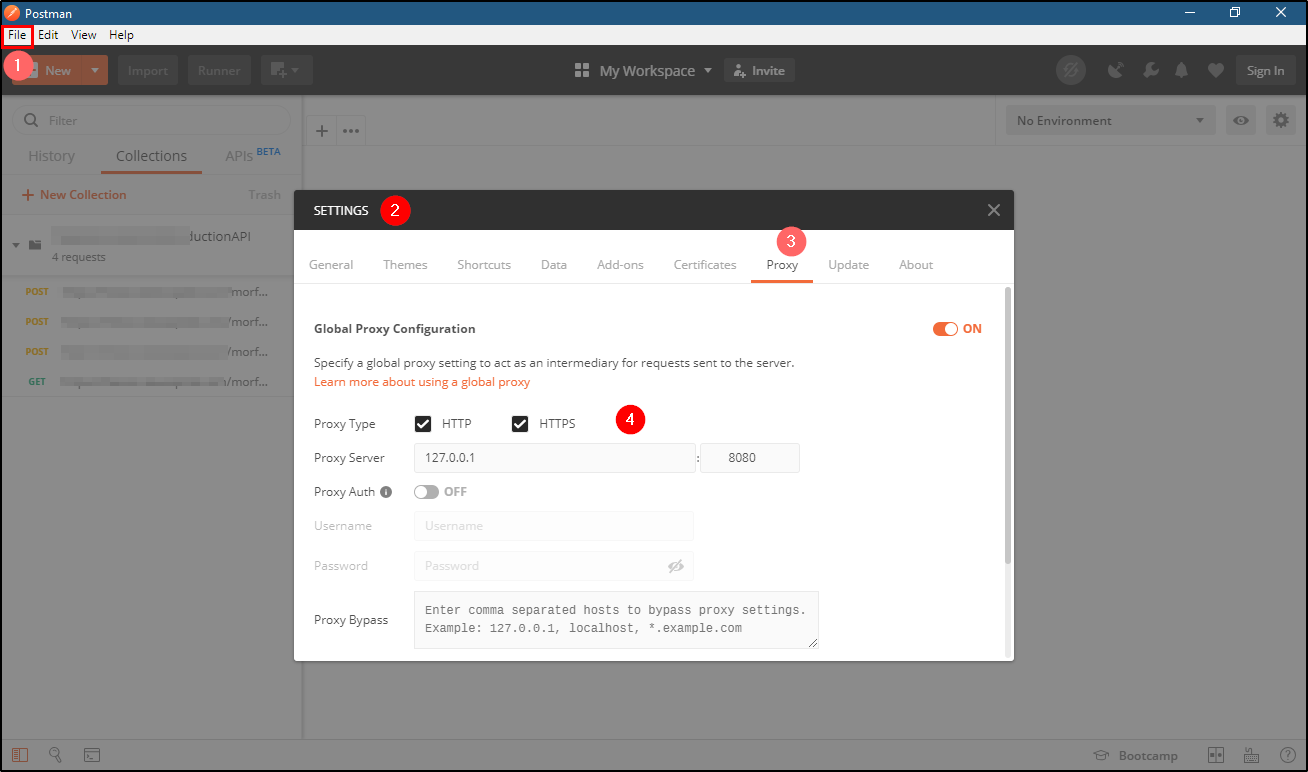
**POSTMAN**: <https://www.getpostman.com/downloads/>

Once you download, it’s as simple as installing normal applications. Please find the screenshots below for POSTMAN. It will look like this. You can log in with your mail or you can skip it also.

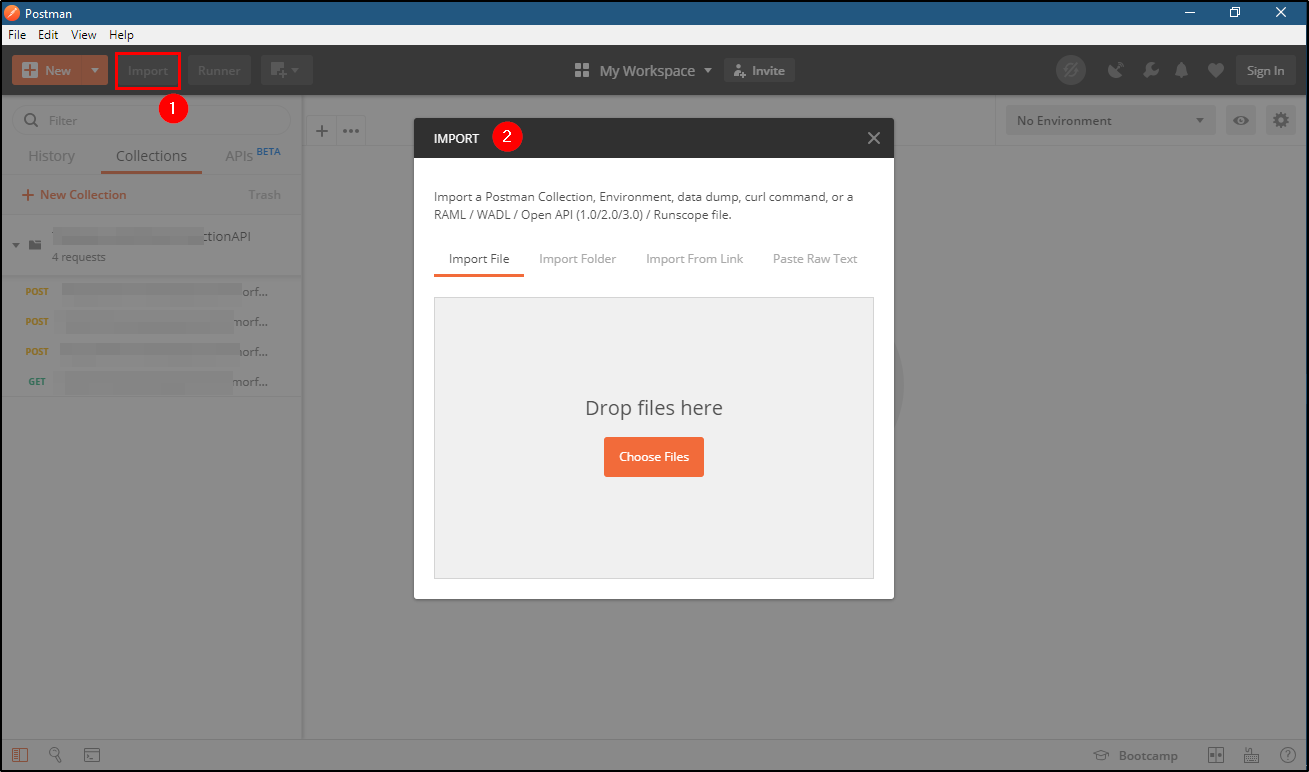


****

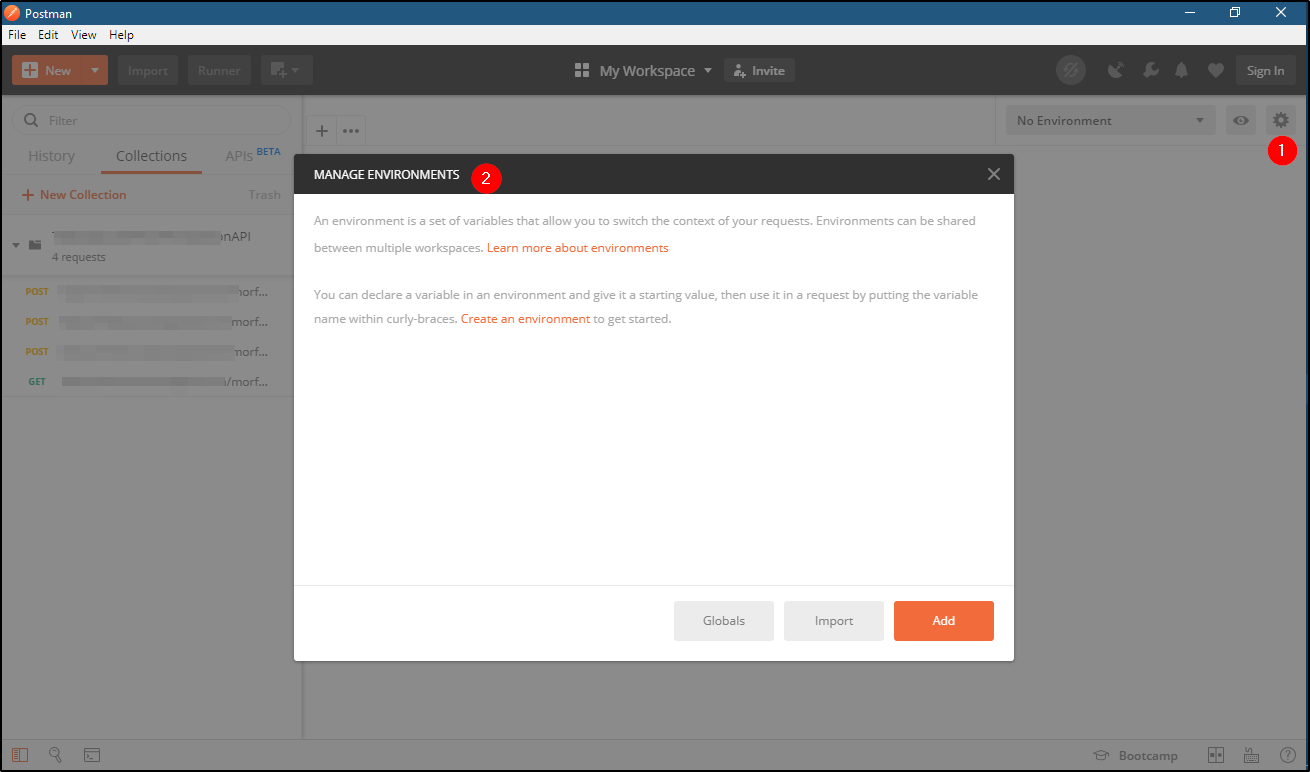
**Now you can go to the FILE tab and then SETTINGS, there you can set up your proxy to take up your all requests in Burp for ease of testing.**

****

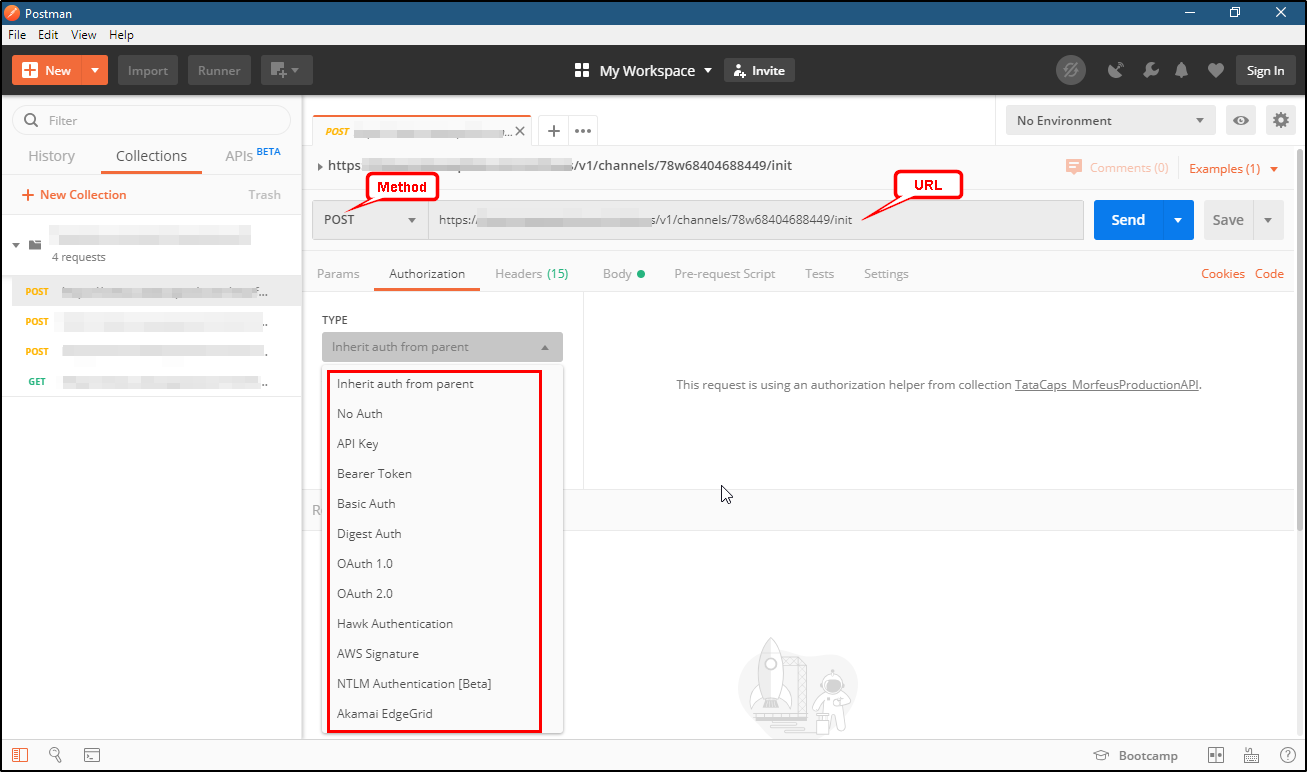
You can simply import API calls and environment, which will save you a lot of manual work.

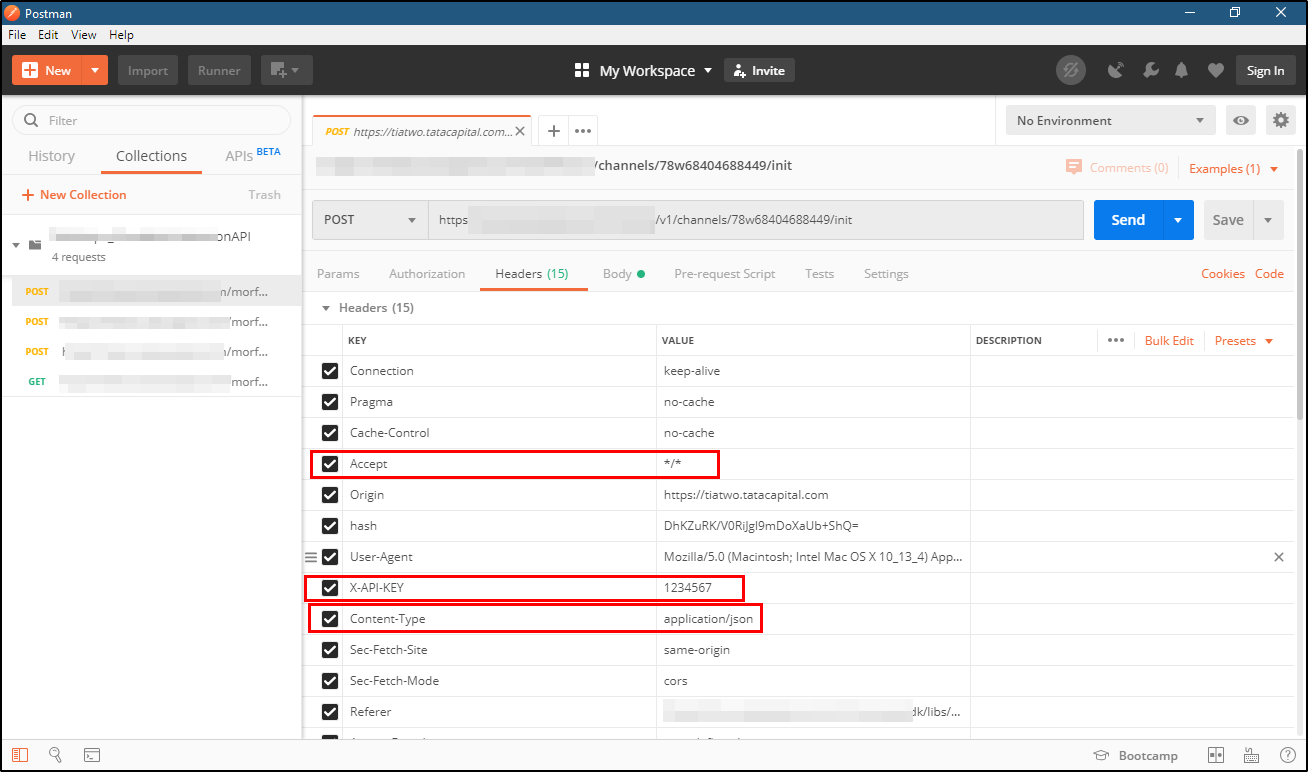


Setting up environment variables:

****

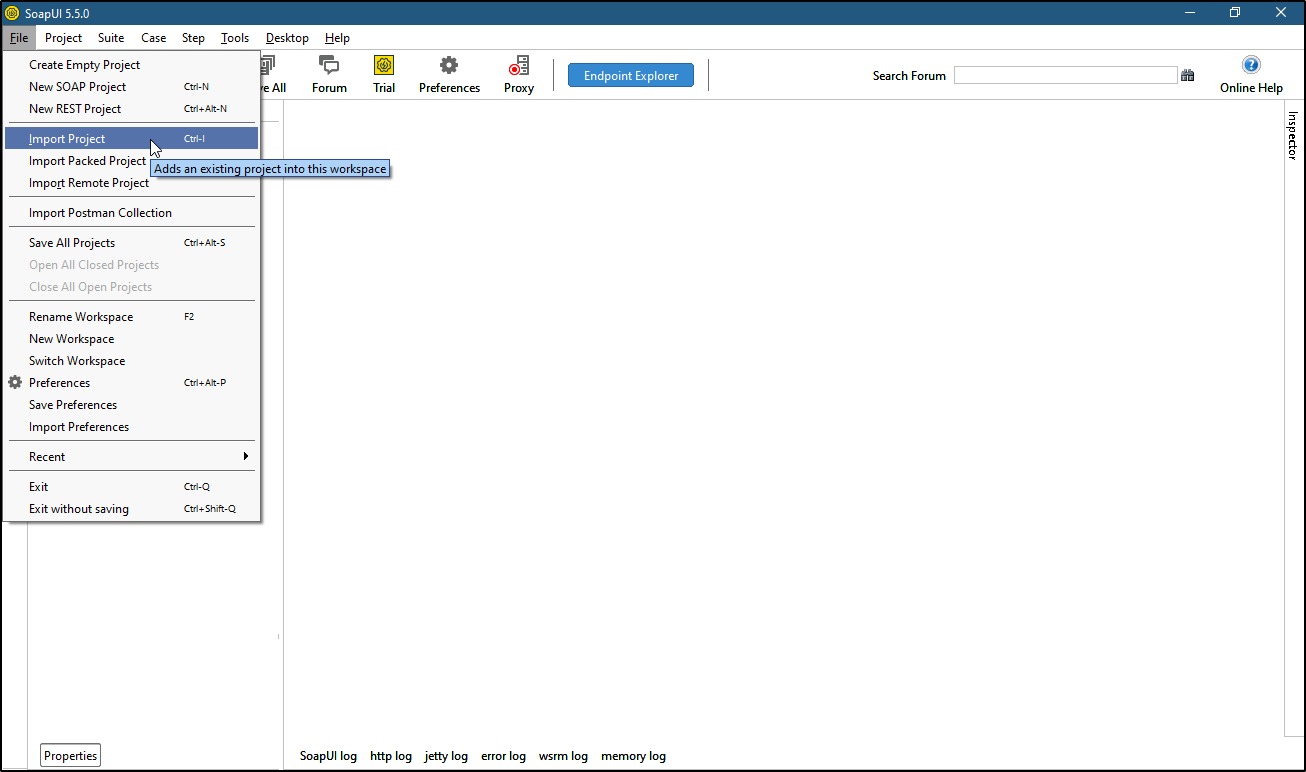
**Authentication Types**

****

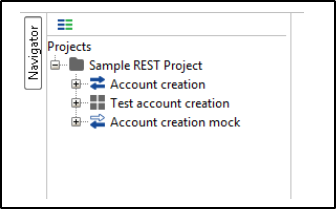
****

**SoapUI**

Importing the Project

****

The sample project will be shown in the SoapUI Navigator.



The structure of a SoapUI project is like this:

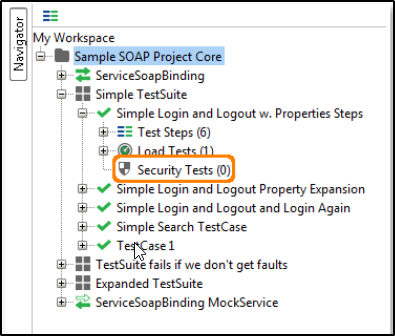
* Project

Interface

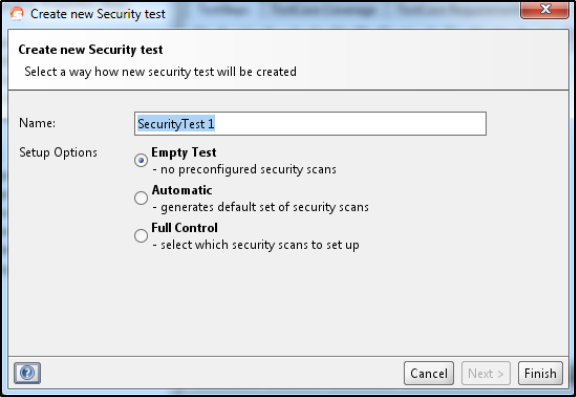
Test suites

Mock services

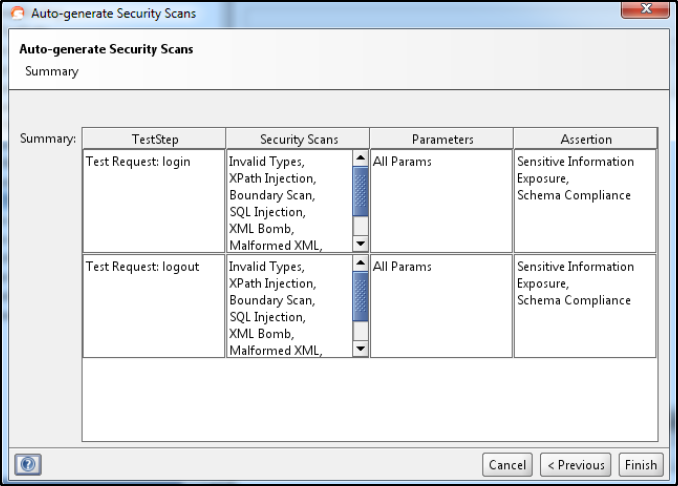
**Create a security testcase**

****

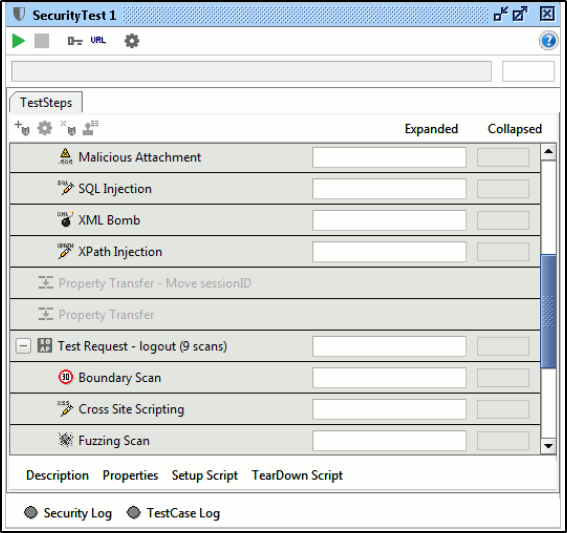
You can see an empty “Security Tests” node in the left tree (see above image), right click it and select the “New SecurityTest” option, this opens the following dialog (if you are using the free version, read further down):

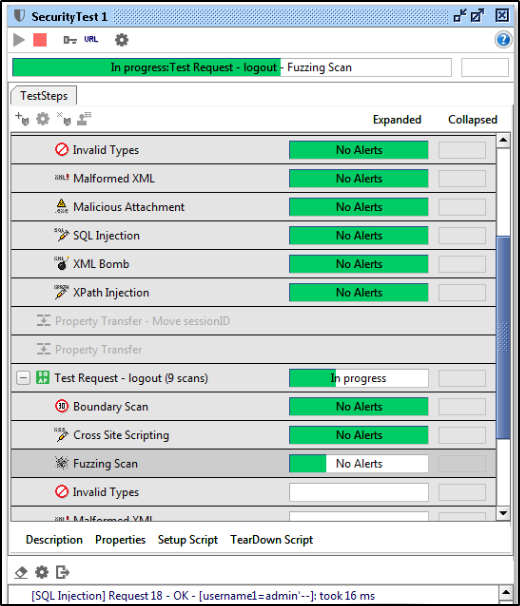
****

Select the “Auto” mode to generate default Security Scans and Assertions for the TestSteps in your TestCase and press "Next":



Here you see a summary of all the Security Scans and Assertions SoapUI will add to the Security Test, press OK to create the Security Test with the described configuration and open the Security Test window:



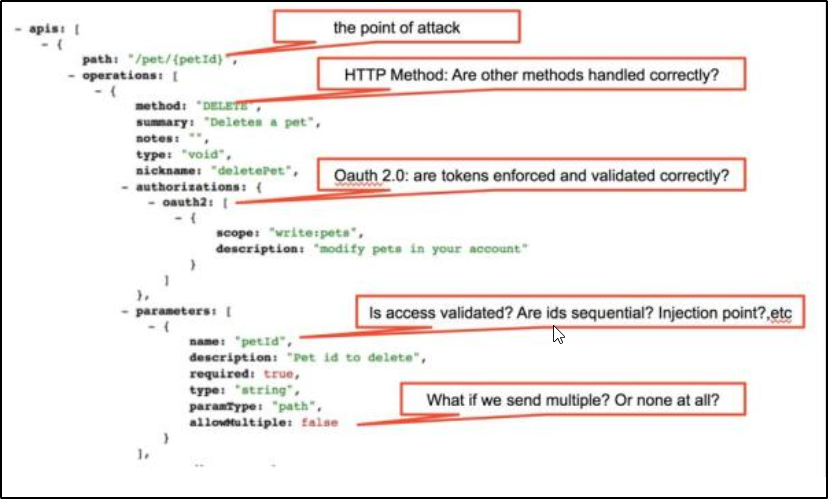


**API Penetration Testing**

**API pentesting** is identical to [web application penetration testing](https://securelayer7.net/en-au/vulnerability-assessment-and-penetration-testing) methodology with some small changes in the attack.

**Approach:**

* Do not jump to testing by getting an end point or set of end points.
* Ask for documentation
* Ask for sample request/response or postman collection.
* Ask for any particular header needed.
* Ask for token or any specific parameter or values for a parameter (to get in right flow))
* Ask for the workflows.
* **API Metadata** and documentation has a lot of momentum currently; API providers are putting strong efforts into providing API consumers with detailed technical descriptions of an API, including all we need for our attack - paths, parameters, message formats, etc. Several standards are at play:
  + Swagger, RAML, API-Blueprint, I/O Docs, etc for REST APIs
  + WSDL/XML-Schema for SOAP APIs

Have a look at the following Swagger definition for example:

As you can see, a helpful Swagger specification also tells us a lot about an API’s possible vulnerabilities, helping us target the attack.

**API Discovery**: what if you have no metadata for the API you want to compromise? An alternative to getting an initial attack surface is to record interactions with the API using an existing client. For example, you might have an API consumed by a mobile app; set up a local recording proxy (there are several free options available) and direct your mobile phone to use this proxy when accessing the API – all calls will be recorded and give you an understanding of the APIs usage (paths, parameters, etc).

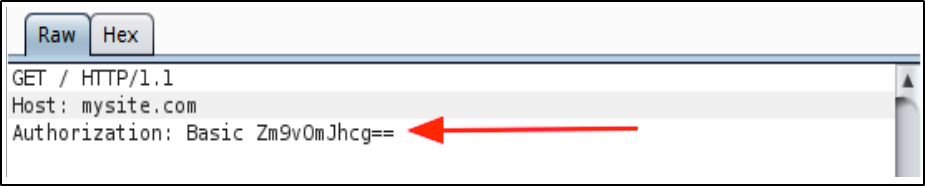
**Brute Force**: full disclosure: most developers aren’t famed for their creativity when deciding on API paths, arguments, etc. More often than not, you can guess at an API’s paths like /api, /api/v1, /apis.json, etc. – which might at least give you something to start with.

**API Authentication:**

**Basic HTTP Authentication**

Basic Auth only requires a username and password.

Base64encode(username:password) = Zm9v0mJhcg==

Passes this value in HTTP header called Authorization. 

When the server receives the request, it looks at the Authorization header and compares it to the credentials it has stored. If the username and password match one of the users in the server's list, the server fulfils the client's request as that user. If there is no match, the server returns a special status code (401) to let the client know that authentication failed, and the request is denied.

Though Basic Auth is a valid authentication scheme, the fact that it uses same username and password to access the API and manage the account is not ideal. That is like a hotel handing a guest the keys to the whole building rather than to a room.

**API Key**

A unique generated value is assigned to each first-time user, signifying that the user is known. When the user attempts to re-enter the system, their unique key is used to prove that they’re the same user as before.

Unlike Basic Auth, which is an established standard with strict rules, API keys were conceived at multiple companies in the early days of the web. As a result, API key authentication is a bit like the wild west; everybody has their own way of doing it; eg : generated from their hardware combination and IP data, and other times randomly generated by the server which knows them. Over time, however, a few common approaches have emerged.

1. The client put the key in the Authorization header, in instead of a username and password.
2. Add the key onto the URL http://example.com?api\_key=my\_secret\_key).
3. Put the key somewhere in the request body next to the data.
4. Custom Header

**Check List:**

* Don't use Basic Auth Use standard authentication (e.g. JWT, OAuth).
* Identify if the API has any authorization token if it is having then remove that authorization token and see application response. In some cases, if authorization is not implemented correctly then API might give you access to forbidden assets of application.
* Don't reinvent the wheel in Authentication, token generating, password storing use the standards.
* JWT (JSON Web Token): Use random complicated key (JWT Secret) to make brute forcing token very hard. Don't extract the algorithm from the payload. Force algorithm in the backend (HS256 or RS256).
* Make token expiration (TTL, RTTL) as short as possible. Don’t store sensitive data in the JWT payload, it can be decoded easily.
* OAuth: Always validate redirect\_uri on server side to allow only whitelisted URLs.
* Always try to exchange for code not tokens (don't allow response\_type=token).
* Use state parameter with a random hash to prevent CSRF on OAuth authentication process.
* Define default scope and validate scope parameter for each application.

Access

* Limit requests (Throttling) to avoid DDoS / brute-force attacks.
* Use HTTPS on server side to avoid MITM (Man in the Middle Attack).
* Use HSTS header with SSL to avoid SSL Strip attack.
* Analyze and check each module with a different access level of user ex: admin, moderator, normal user.
* Check whether admin modules can be accessed via the restricted user.

Input Validation

* An API should provide expected output for a given input
* The inputs should appear within a range and values crossing the range must be rejected
* Any empty or null input must be rejected when it is unacceptable
* Incorrectly sized input must be rejected
* Check injection vulnerabilities by inserting special characters in all parameters in a request and check the response from the server. If you find any stack traces, then use the information for further exploitation.
* Insert greater than, less than (<,>) characters in all parameters and see response whether the application encoding them as > and <. If an application doesn’t escape any special characters then the application may be vulnerable to client-side attacks such as XSS (cross-site scripting).

Parameter Tampering

* It takes the advantage of backend sanitizing errors and then manipulates parameters sent in API requests.
* According to this, the forms that use type=”hidden” input should always be tested in order to make sure that backend server correctly validates them.

<input type=”hidden” name=”price” value=”100.00″ />

Conceptually, when the user opens his web browser and changes the input valued from 100.00 to 1.00 and submit the form, then the service will be vulnerable to parameter tampering.

* Observe each parameter in every module of API, understand how the data is transferred from source to destination. Try to play with the parameter by tampering them.
* Identify the parameters which may vulnerable to IDOR type vulnerabilities such as id=1234 and also look at the cookies for manipulating the Ids.
* Modify the content-type server header for understanding the XML entity injection attack. Example: change content Application/JSON to application/XML and insert the XML entity payload to find the XML entity injection.

Processing

* Check if all the endpoints are protected behind authentication to avoid broken authentication process.
* User own resource ID should be avoided. Use /me/orders instead of /user/654321/orders.
* Don't auto-increment IDs. Use UUID instead.
* If you are parsing XML files, make sure entity parsing is not enabled to avoid XXE (XML external entity attack).
* If you are parsing XML files, make sure entity expansion is not enabled to avoid Billion Laughs/XML bomb via exponential entity expansion attack.
* Use a CDN for file uploads.
* If you are dealing with huge amount of data, use Workers and Queues to process as much as possible in background and return response fast to avoid HTTP Blocking.
* Do not forget to turn the DEBUG mode OFF.

Output

* Send X-Content-Type-Options: nosniff header.
* Send X-Frame-Options: deny header.
* Send Content-Security-Policy: default-src 'none' header.
* Remove fingerprinting headers - X-Powered-By, Server, X-AspNet-Version, etc.
* Force content-type for your response. If you return application/json, then your content-type response is application/json.
* Don't return sensitive data like credentials, Passwords, or security tokens.
* Return the proper status code according to the operation completed. (e.g. 200 OK, 400 Bad Request, 401 Unauthorized, 405 Method Not Allowed, etc.).

Rate Limiting

* Especially important if your API is public-facing so your API and back-end are not easily DOSed. Simple rate limits are available in many web servers and proxies, though more sophisticated entity intensity tracking is even better.

Rate limiting prevents malicious code from abusing legitimate or illegitimate access to the API.

HTTP/1.1 429 Too Many Requests

Content-Type: text/html

Retry-After: 3600

Geofencing

* If your API is public, it might make sense to either block users from countries you don't do business with, or at least raise the risk score of entities that come from those countries