**Application Programming Interface (API) Testing**

An application programming interface is a set of subroutine definitions, communication protocols and tools for building software. Generally, it is a set of clearly defined methods of communication among various components. It is a software intermediary that allows two applications to talk to each other. Each time you use an app like Facebook, send an instant message, or check the weather on your phone, an API is being used. A set of procedures, functions, and other points of access which an application, an operating system, a library etc., makes available to programmers in order to allow it to interact with other software. A good API makes it easier to develop a computer program by providing the building blocks, which are then put together by the programmer.

An API may be for a web-based system, operating system, database system, computer hardware, or software library.

**What Is an Example of an API?**

When you use an application on your mobile phone, the application connects to the Internet and sends data to a server. The server then retrieves that data, interprets it, performs the necessary actions and sends it back to your phone. The application then interprets that data and presents you with the information you wanted in a readable way. This is what an API is - all of this happens via API.

To explain this better, 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. What of if the dish is not available? Well waiter knows beforehand that you have made the wrong request, so he will tell you then and there on the table that the food item is not available. How much time and energy is saved? This is exactly what an API does.

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 Kayak or Expedia, 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.

In summary, an API takes a request and gives a response verifying that the response matches the request and that null value is not returned.

## What an API Also Provides Is a Layer of Security

Your phone’s data is never fully exposed to the server, and likewise the server is never fully exposed to your phone. Instead, each communicates with small packets of data, sharing only that which is necessary—like ordering takeout. You tell the restaurant what you would like to eat, they tell you what they need in return and then, in the end, you get your meal.

APIs have become so valuable that they comprise a large part of many business’ revenue. Major companies like Google, eBay, Salesforce.com, Amazon, and Expedia are just a few of the companies that make money from their APIs. What the “[API economy](https://www.mulesoft.com/webinars/api/welcome-api-economy)” refers to is this marketplace of APIs.

## The Modern API

Over the years, what an “API” is has often described any sort of generic connectivity interface to an application. More recently, however, the modern API has taken on some characteristics that make them extraordinarily valuable and useful:

* Modern APIs adhere to standards (typically HTTP and REST), that are developer-friendly, easily accessible and understood broadly
* They are treated more like products than code. They are designed for consumption for specific audiences (e.g., mobile developers), they are documented, and they are versioned in a way that users can have certain expectations of its maintenance and lifecycle.
* Because they are much more standardized, they have a much stronger discipline for security and governance, as well as monitored and managed for performance and scale
* As any other piece of productized software, the modern API has its own software development lifecycle (SDLC) of designing, testing, building, managing, and versioning.  Also, modern APIs are well documented for consumption and versioning.

**What is an API Testing?**

The API testing is performed for the system, which has a collection of APIs that ought to be tested. During testing, a test of the following things is looked at.

* Exploring boundary conditions and ensuring that the test harness varies parameters of the API calls in ways that verify functionality and expose failures.
* Generating more value-added parameter combinations to verify the calls with two or more parameters.
* Verifying the behaviour of the API which is considering the external environment conditions such as files, peripheral devices, and so forth.
* Verifying the sequence of API calls and check if the API produce useful results from successive calls.

**Common Tests Performed on API’s**

* Return Value based on input condition – The return value from the API’s are checked based on the input condition.
* Verify if the API’s does not return anything.

Why Should We Be Using API Testing?

Many companies are moving toward a ***microservices*** model for their software applications. This means that different sections of their application can have separate datastores and separate commands for interacting with the datastore. Microservices are appealing to software provider because they allow components of the software to be deployed more quickly; while one area of an application is updated, the other areas of the application can continue to function.

We live in a time when users expect software to be ready to use whenever they want it, and the microservices model ensures that an application will not be down for maintenance when a user wants to use it.

Most microservices are using application programming interfaces (APIs), which are a set of commands for how a service can be Representational State Transfer (REST) request through Hypertext Transfer Protocol (HTTP) to request and send data.

While some UI testing will always be needed to verify that elements appear on a page and can be interacted with, API tests are much faster and more reliable than UI-based tests. When testing things like adding and removing records from a database, automated UI tests can be time-consuming and repetitive. API tests allow for rapid record manipulation.

How Does a REST Request Work?

An API is made up of a set of REST requests. These are the requests made to an application server that retrieve, delete, or manipulate data in an application’s database.

A REST request is made up of the following parts:

* An HTTP verb that describes what action should be taken
* A Uniform Resource Locator (URL) that defines the location of the request
* HTTP headers that provide information to the server about the request
* A request body that provides further details for the request ( this can sometimes be empty)

Here are the most common HTTP verbs:

* A GET request fetches a record from a database
* A POST request adds a new record to a database
* A PUT request replaces a record with a new one
* A PATCH request replaces part of a record with new information
* A DELETE request removes a record from a database

The URL used in the request clarifies which type of record will be altered by the request. For example, a GET request used in conjunction with the URL <https://www.example.com/cars/1> would return record number 1 in a table of cars.

Similarly, a GET request used in conjunction with the URL <https://www.example.com/trees/2> would return record the second record in a table of trees. The end of the URL specifies the **endpoint** – a data object used in the API. In these examples, the endpoints are /cars and /trees.

HTTP headers can provide information to the server such as :

* The **Host**: the domain and port number of the user making the request
* **Authorisation**: the credentials of the user making the request
* The **Content-type**: the format of the information provided in the body of the request

The Request body is used one making POST, PUT or PATCH request. The body specifies exactly what information should be added to the database. It is usually in JavaScript Object Notation (JSON) or Extensible Markup Language(XML) format. Here is an example of what a JSON request body might look like for doing a POST request that adds a customer to a database:

{

“firstName”:”John”

“lastName”:”Smith”

“emailAddress”:”jsmith@example.com”

}

**What is Included in the Response to a REST Request?**

The response to a REST request is the information that the server sends back after it has received and processed the request. It will include:

* HTTP headers that describe the response
* A response code that describes the success or failure of the response and
* A response body that includes requested or relevant information(this can sometimes be empty)

The HTTP headers in the response provide information to the requesting party such as :

* **Access-Control headers**: tell the requester what types of requests and headers will be allowed.
* **The Content-Type**: the format of the information returned in the response
* **The Server**: the name of the server that responded to the request

Response codes are three-digit codes used to describe the result of the REST request. The most common response codes come in one of these three categories:

* ***200-level*** responses indicate that the request was received, understood, and processed.
* ***400-level*** responses indicate that the request was received, but that there was an error from the client
* ***500-level*** responses indicate that there was some sort of server error

The response body will vary by what type of request was made and whether there was an error in the request.

A successful GET request will return a body that has the information requested. A successful POST request might return a body that echoes the body sent with the POST, or it might return an empty body with a 200-level response code. A request that returns an error may have an error message in the body of the response.

HTTP Status Codes

Code Title Description

200 OK The request was successful

201 Created The resource was successfully created

202 Async created The resource was asynchronously created

400 Bad request Bad request

401 Unauthorised Your API key is invalid

402 Over quota Over plan quota on this endpoint

404 Not found The resource does not exist

422 Validation error A validation error occurred

50X Internal Server Error

**How Can We Test APIs?**

Now that the mechanics of how a REST request work is known, the main quest is how they can be tested. The first step in testing an API is to determine what sort of REST request can be made by the API and what limitations are for each request. This can be done by looking at the documentation.

Some developers may choose to document the API by using ***Swagger***, which is open-source framework that clearly describes an API’s behaviour. The re are other documentation tools available as well.

In other to test an API completely, it is important to understand what the limitations are for each REST request in the API. Finding, testing, and verifying those limits is crucial to making sure that the API will behave in a way that users expect, and that the API cannot be exploited by malicious users to get information they should not have access to.

Here are some questions to ask the API developer when collecting information about an API.

* What endpoints are available?
* What HTTP verbs can be used with the endpoints?
* Are any of the verbs limited by authorisation?
* Which fields are required in the requests?
* What are the validation limits on the fields?
* What response codes should I expect for a successful request?
* What response code should I expect for an Unsuccessful request?
* What sort of error messages will be returned in the body of an unsuccessful request?

The next step is to set up each of these requests in an API test tool.

A variety of tools can be used for this purpose. It is also possible to write API tests directly into code, but the advantage of API testing tools is that they are easy to use and provide a way to visualise the response.

The easiest API testing tool to use is Postman. It has a 100 percent free version that can be downloaded quickly, and there are also paid versions for teams.

Runscope and SOAPUI are two other popular tools.

Once you have chosen an API testing tool, the first requests to set up are the “happy path” requests. These are the requests that the API developer would expect that users would make in the normal course of using the application.

When setting up a “happy path” – style request, it is important to include assertions. One assertion should be that the correct response code is returned(often a 200 response).

If the response includes a body, there should be an assertion on that as well. For example , if a GET request is being tested, there should be an assertion that the body of the response contains the data that was expected with that record.

Once all the “happy path” tests have been created, negative tests can be added. Negative tests are those that make sure that any kind of error is handled correctly. It is important that an application doesn’t crash when a user accidentally imports invalid data, and it is important that a malicious user is prohibited from entering harmful scripts into the database. Here are some examples of negative tests:

* Sending a request with the wring HTTP verb
* Sending a request with the wrong endpoint
* Sending a request with the wrong header
* Sending a request with missing header
* Sending a request without authorisation
* Requesting data for a record that does not exist
* Sending a request with a body that has missing required fields
* Sending a request with a body that has invalid field values

Walkthrough Example of API Testing

The developers of Swagger UI have created an API to practice on, called the Swagger Petstore. This is the only public practice API I know of that provides a clear visualisation of how each API request works, and also provides a way to try out requests directly in the documentation. We can create a request that points to this API in Postman, add some assertions, and run if from the command line with Newman. Newman is a command-line run tool created by Postman that can be installed through the node package manager(npm).

**Step1**

* Navigate to <https://petstore.swagger.io> and click on the request “GET /pet/{petId}” to open it.
* Click the “Try it out” button.
* Put the number 1 in the field marked “ID of pet to return”
* Click the “Execute” button
* Scroll down to the Response body and verify that a record for a pet has been returned (if a record was not returned, try the request again with a different ID in the “ID of pet to return” field

**Step 2**

* Download Postman by going to https://www.getpostman.com and clicking the “Download the App” button

**Step 3**

* Launch Postman.
* In the top middle of the screen, verify that you see the word “GET” in the HTTP verb dropdown.
* In the space beside the “GET” , enter this URL: <http://petstore.swagger.io/v2//pet/1> (if you are using a different ID than 1 , replace the 1 with the ID you chose).
* Click the “Send” button and verify that you receive a response code of 200 OK, and that the response body contains the record for the pet.

**Step 4**

* Click on the “Tests” tab found beneath the URL of the Postman request
* On the right of the screen, scroll through the selection of code snippets until you find the one titled “Status code: Code is 200,” and click on this title.
* Note that this assertion has been added to the test window. The code asserts that 200 is the response code you get when you send the request.

**Step 5**

* Scroll through the selection of code snippets until you find the one titled “Response Body: JSON value check,” and click on this title.
* Note that this assertion has been added to the test window.
* Change the string ”Your test name” to “Correct pet ID is returned”; this will be the name of the test.
* Change the value in parentheses after the word “expect” from “jsonData.value” to “jsonData.id”; the test will be checking on the value of the pet ID.
* Change the value in parentheses after the word “eql” from “100” to “1” (or whatever ID you chose to use in the GET request).
* Click the “Send” button again, and click the “Test Results” tab.
* Verify you see the two tests you created with the word “Pass” next to
* Open a command window, type “npm install -g newman” and click the Return key. This will install Newman onto your machine. If you are new to using the command line, see this article for some of the common commands.
* Using the command window, navigate to the location where you saved the JSON file (for example, you might type cd Desktop to change to the desktop folder).
* Type this command into the command window: ”newman run PetStore.postman\_collection.json”
* You should see your tests run and pass in the command window.

**How Can We Automate API Testing?**

Once a complete suite of positive and negative tests has been created, automation can be set up. The Newman command to run the tests can be integrated into continuous integration (CI) project. Similarly, Runscope and SoapUI also provide command-line functionality for integration with CI tools.

Next Steps

The above walkthrough is just a simple example of what is possible with API testing. To go further. You can try out more of the requests at the swagger Petstore and create tests for them in Postman.

Take a look at the API in your company to find out what kinds of automated APII tests you could set up for your software.

Examine your automated UI tests to see which ones might be better covered by API testing. For example If you have a UI test that fills out a customer form and add a new record to the database, you could instead have an API test the uses a POST request to add the record. API test are best for testing the creation and manipulation of data, while UI test are best for checking the onscreen availability and usability of buttons and links.

Moving the bulk of your automated tests to API testing we allow your test suite to run much faster and more reliably. You will be able to quickly uncover flaws in the manipulation of the data store and expose any potential security holes before they can be exploited.