1. **The main objectives in functional testing of the API are:**

- to make sure the implementation is operating as it should – no bugs!
- to confirm that the implementation is performing as expected in accordance with the requirements (which later on becomes our API documentation).

The API test strategy provides information on individual test scenarios and cases. Below are three core objectives of API functional testing.

- APIs Ensure Proper Implementation: Complex coding in software development can occasionally result in errors.
- They make sure that the executions go as planned: This API method will become your documentation in the future. It offers information on the lifecycle of an API, such as new versions, upgrades, and constraints.
- They Provide Regressions Between Code Mergers and Releases: The objective of any file is to merge its algorithms with other programs without conflict. However, changes may sometimes cause friction. API enables developers to enhance features seamlessly.

API Test Actions

One API test is made up of several test actions, making them vital in a test flow. The following should be included in every API request:

Verify the HTTP Status Code: This is a critical step in swiftly identifying and correcting broken links. Unauthorized requests, such as redirections, client errors, and server failures, can be returned using a resource.

Verify Response Headers: HTTP requests that aren't related to the message are known as response headers. To acquire more context about an answer, you can use factors like age, location, or server.

Verify Response Payloads: This action will help you use JSON data in addition to allowing you to see responses from curl or Postman. Field names, types, values, and errors should all be included.

Examine the Basic Performance Sanity testing ensures that the system is bug-free and that changes to the code will not be affected. It also examines the speed with which any test is completed.

Verify Application State: Although this operation is optional, we strongly advise it. It can be used for manual testing or when you have access to a user interface (UI).

Check that APIs Follow Proper Security Protocols

Performance Tests

Because of its functions, it’s essential for APIs to be accurate, secure, and reliable.

Load and Stress Tests

Load tests are used by software developers to evaluate system performance under real-world load scenarios.

Usability Tests

Usability tests, often known as user experience testing, are used to assess how user-friendly a piece of software is. Experts use them to find and remedy flaws that obstruct smooth consumer operations.

2. Using API testing tools is how development and testing teams qualify software performance before they get to the users’ hands. Now guided by the principles of Agile, the main benefits QAs and developers are after include:

Earlier testing and feedback without waiting for the whole software to be completely built

Faster test maintenance and refactoring

More bugs and issues were spotted in less time compared to UI tests

- **POSTMAN** – It will be used since it is a Google Chrome plugin that can be used to test API services. It's a robust HTTP client for evaluating online services. Postman is an excellent alternative for manual or exploratory API testing.
- With Postman, almost any data from a modern web API can be pulled. Within the Postman Interface, you can write Boolean tests. You can build a collection of REST calls and preserve each one as a collection for later execution. It is not a command line-based tool, unlike CURL, which eliminates the need to paste text into a command line window. Postman is more dependable when it comes to sending and receiving REST data.

- Selenium - Open-source automation testing tools for web applications are included in the list of open-source automation testing tools for web apps. This is a tester's all-time favourite tool.
- Appium is considerably one of the best mobile app testing tools used by most professional testers.
- Jmeter Web dynamic apps are used to test performance on both static and dynamic resources.
- SQL TEST Last, but not least, we have the SQL Test. Using its SQL server management studio, this program allows you to build unit tests for SQL server databases.

3. Test Frequency

As soon as a developer pushes a commit, our Jenkins run unit tests, then integration tests (e.g. just backend) and finally system tests (e.g the whole app: backend, frontend, platforms, etc). The system tests stage is fairly quick, about 3 minutes. Then, once a day, I will run stability tests for 2 hours.

4.

There are four main types of API testing methods as follows −

- **GET –** This function is used to retrieve data from a server using a URI (Uniform Resource Identifier). This method should only be used to extract data and should not alter it in any manner.
- **POST –** This technique is used to create new entities as well as submit data to the server via HTML forms, such as customer information, file uploads, and so on.
- **PUT –** This method is used to update an entity or create a new one.
- **DELETE –** This technique is used to remove any existing representations of a URI's target resource.

First up, I want to come up with a test plan so I can get good test coverage. This is called test selection criteria. There are lots of different approaches to coming up with my tests and in this example, I will be going to look at using a requirements-based criteria plus a data coverage criterion to come up with something pretty solid.

Positive tests are scenarios that examine a user flow that does not result in any faults, i.e. something that the user can complete without encountering any issues. Negative tests are scenarios that test a user flow that is expected to create problems. If a user enters a necessary field, for example, an error notice occurs.

The positive test cases are the regular user flows that are expected:

- POST- Start the web Application/Mobile App, click the create button, input form values, click the save button.
- GET- Start the web Application/ Mobile App, select an entity, click the view button.
- PUT - Start the web Application/ Mobile App, select an entity, click the update button, change form values, click the save button.
- DELETE- Start the web Application/ Mobile App, select an entity, click the delete button, click the ok button.

REST API	**DATABASE/SQL**	**HTML/XML**
POST	new row, no change to other rows	list page, new entity in list

GET	new row, no change to other rows	entity page, entity values displayed
PUT	change to this row only	list page, updated entity in list
DELETE	deleted row, no change to other rows	list page, entity deleted from list

Negative Tests

The negative test cases are the irregular user flows that are expected:

- POST - Start the Web/Mobile Application click the create button, input incorrect form values, click the save button.
- POST - Start the Web/Mobile Application, click the create button, input correct form values, click the cancel button.
- GET - Start the Web/Mobile Application, do not select an entity, click the view button.
- PUT - Start the Web/Mobile Application, do not select an entity, click the update button.
- PUT - Start the Web/Mobile Application, select an entity, click the update button, change to incorrect form values, click the save button.
- PUT - Start the Web/Mobile Application, select an entity, click the update button, change to correct form values, click the cancel button.
- DELETE - Start the Web/Mobile Application do not select an entity, click the delete button.
- DELETE - Start the Web/Mobile Application select an entity, click the delete button, click the cancel button.

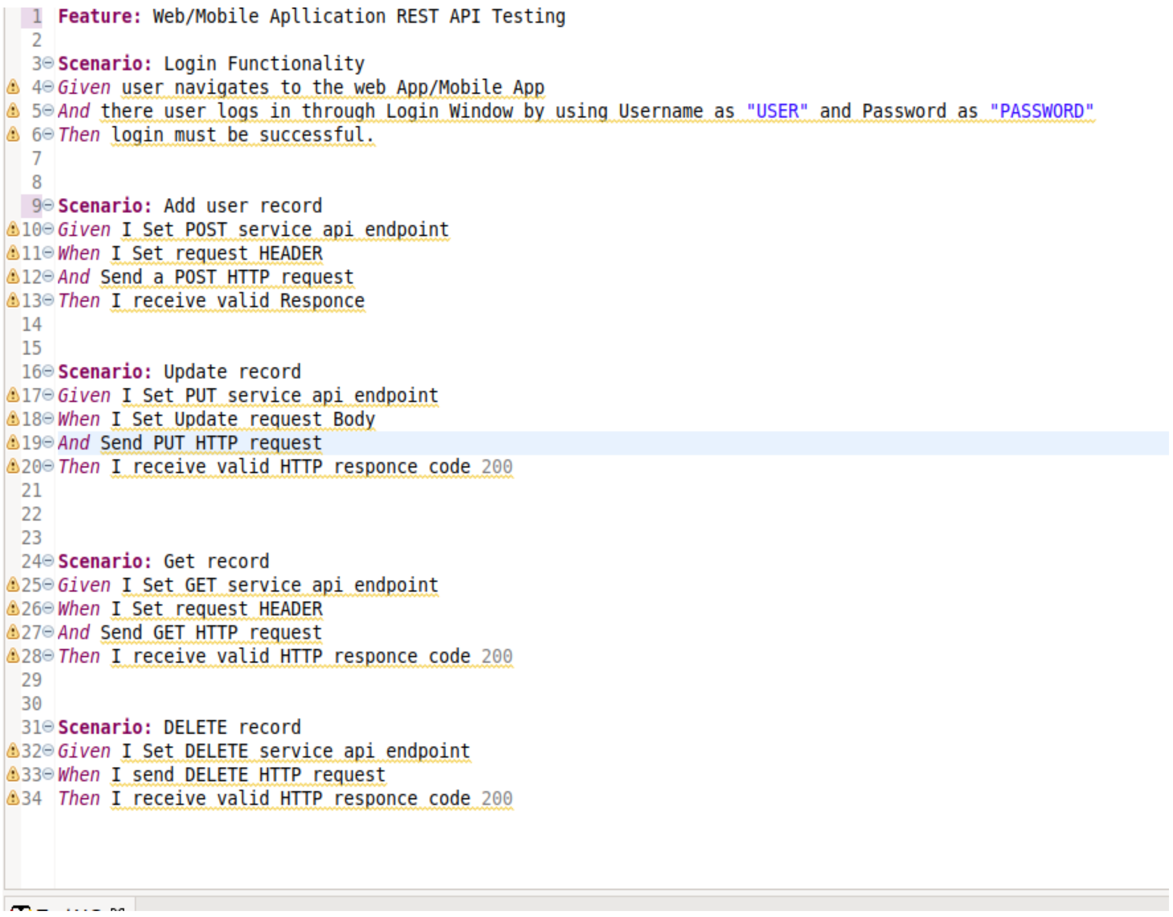
REST API	**DATABASE**	**HTML/XML**
POST	no changes	error message displayed
POST	no changes	no changes
GET	no changes	no changes

PUT	no changes	no changes, error message displayed
PUT	no changes	error message displayed
PUT	no changes	no changes
DELETE	no changes	error message displayed
DELETE	no changes	no changes

TEST CASES

Now that I have a test plan, the next step is to write some tests.

I'll require browser automation solutions to test web-based systems like the Web/Mobile application. Selenium is one of the most popular, with a large number of programmable utilities. I should also explore tools like Cucumber and Appium, which provide a great domain specific language for writing more understandable tests, in addition to Selenium.



```
Feature: Web/Mobile Apllication REST API Testing

Scenario: Login Functionality
Given user navigates to the web App/Mobile App
And there user logs in through Login Window by using Username as "USER" and Password as "PASSWORD"
Then login must be successful.


Scenario: Add user record
Given I Set POST service api endpoint
When I Set request HEADER
And Send a POST HTTP request
Then I receive valid Responce


Scenario: Update record
Given I Set PUT service api endpoint
When I Set Update request Body
And Send PUT HTTP request
Then I receive valid HTTP responce code 200



Scenario: Get record
Given I Set GET service api endpoint
When I Set request HEADER
And Send GET HTTP request
Then I receive valid HTTP responce code 200


Scenario: DELETE record
Given I Set DELETE service api endpoint
When I send DELETE HTTP request
Then I receive valid HTTP responce code 200
```

Step Definition

Steps definition file stores the mapping between each step of the scenario defined in the feature file with a code of function to be executed.

```
package my.package.name


import cucumber.api.PendingException;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.And;

import cucumber.api.junit.Cucumber;

import org.junit.runner.RunWith;


@RunWith(Cucumber.class)

public class MyStepDefinitions {


    @Given("^user navigates to the web App\/Mobile App$")

    public void user_navigates_to_the_web_appmobile_app() throws Throwable {    }

    @Given("^I Set POST service api endpoint$")

    public void i_set_post_service_api_endpoint() throws Throwable {    }

    @Given("^I Set PUT service api endpoint$")

    public void i_set_put_service_api_endpoint() throws Throwable {    }

    @Given("^I Set GET service api endpoint$")

    public void i_set_get_service_api_endpoint() throws Throwable {    }

    @Given("^I Set DELETE service api endpoint$")
    public void i_set_delete_service_api_endpoint() throws Throwable {    }

    @When("^I Set request HEADER$")

    public void i_set_request_header() throws Throwable {    }

    @When("^I Set Update request Body$")

    public void i_set_update_request_body() throws Throwable {    }


    @When("^I send DELETE HTTP request$")

    public void i_send_delete_http_request() throws Throwable {    }

     @Then("^login must be successful\.$")

    public void login_must_be_successful() throws Throwable {    }


    @Then("^I receive valid Responce$")

    public void i_receive_valid_responce() throws Throwable {    }


    @Then("^I receive valid HTTP responce code 200$")

    public void i_receive_valid_http_responce_code_200() throws Throwable {    }

    @And("^there user logs in through Login Window by using Username as \"([^\"]*)\" and
Password as \"([^\"]*)\"$")

    public void
there_user_logs_in_through_login_window_by_using_username_as_something_and_passwo
rd_as_something(String strArg1, String strArg2) throws Throwable {    }


    @And("^Send a POST HTTP request$")

    public void send_a_post_http_request() throws Throwable {    }
```

```
   @And("^Send PUT HTTP request $")

   public void send_put_http_request() throws Throwable {    }


   @And("^Send GET HTTP request$")

   public void send_get_http_request() throws Throwable {    }

}
```

SELENIUM

WebDriver

```
System.setProperty("webdriver.chrome.driver","/home/user/Desktop/chromedriver_linux64/chromedriver");

        driver = new ChromeDriver();

        driver.get("https://login");
```

Login code

```
driver.findElement(By.xpath("//input[@type ='text']")).sendKeys(loginEmail);

        driver.findElement(By.xpath("//input[@type ='password']")).sendKeys(password );

        driver.findElement(By.xpath("//button[@type ='submit']")).click();

        Thread.sleep(2000);


driver.findElement(By.xpath("//*[@id=\"root\"]/div[1]/div[2]/div[1]/div[1]/div[2]/div[1]/div[1]/nav[1]/ul[1]/li[7]/a[1]/span[1]")).click();
```

Click Button To login

```
driver.findElement(By.xpath("//*[@id=\").click();

        Thread.sleep(5000);
```

Test Runner Class*

```
package cucumberTest;

import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(
		features = "Feature"
		,glue={"stepDefinition"}
		)

public class TestRunner {

}
```

Manually Testing

Usability Testing has now become a vital part of any web-based project. It can be carried out by testers like you or a small focus group similar to the target audience of the web application.

Test the site **Navigation**:

- Menus, buttons or Links to different pages on your site should be easily visible and consistent on all webpages

Test the **Content**:

- Content should be legible with no spelling or grammatical errors.
- Images if present should contain an “alt” text