Srinivas Donapati

**REST API testing framework**

**REST API testing framework**

Contents

[1 Introduction 2](#_Toc499480541)

[2 API TESTING 3](#_Toc499480542)

[3 HTTP 4](#_Toc499480543)

[4 Web-API Testing 5](#_Toc499480544)

[5 Framework 6](#_Toc499480545)

[5.1 Property files 7](#_Toc499480546)

[5.2 Properties 8](#_Toc499480547)

[5.4 Test Class 13](#_Toc499480548)

[5.5 Test Suite 15](#_Toc499480549)

[6 Test Configuration 17](#_Toc499480550)

[7 Project Setup and Flow 19](#_Toc499480551)

[8 File Upload requests 20](#_Toc499480552)

[9 File Download requests 22](#_Toc499480553)

[10 Chrome Extension 23](#_Toc499480554)

[11 Resources 24](#_Toc499480555)

# Introduction

This document describes about the new approach for **WEB-API** testing. A typical API/REST testing requires to have a good knowledge of HTTP and JAVA (or any other language). This framework is aimed for writing/creating functionality (API) tests with minimal code.

The following sections will discuss details about the background, framework and implementation.

# API TESTING

API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing. It helps in determine if they meet expectations for functionality, reliability, performance, and security.

Since APIs lack a GUI, API testing is performed at the message layer. API testing is now considered critical for automating testing because APIs now serve as the primary interface to application logic. As GUI tests are difficult to maintain with the short release cycles and frequent changes, API testing is becoming popular in automation.

# HTTP

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting data. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response.

A client can be a web browser, a desktop or a mobile application. This protocol is widely used for transmitting data between applications and servers.

There are 4 different methods to connect.

* GET

The GET method is used for retrieving the read-only data from the server. And it should not be used for write operations.

* POST

The POST method is used for adding/creating new data to the server.

* PUT

The PUT method is used for updating existing data, if the data to be updated is not present it will create new entity for that.

* DELETE

The DELETE method deletes the specified resource.

# Web-API Testing

A typical web server supports data transmitting using **API**s. These APIs can be of Restful, Servlets based or web services. In a cloud integrated applications (Web, Mobile or Desktop) the functionality depends on how the application communicates with the server.

The API automation would be very much helpful in testing the functionality. As it doesn't deal with the UI, there will be less maintenance and less execution time.

# 5 Framework

A typical web-API testing includes

* Setup server connection
* Build request
* Parse response
* Test response

These all steps requires a lot of code to be written. And most of the code would be similar or same for different tests/API calls. As a result the implementation and code maintenance becomes time consuming.

To reduce the complexity and maintenance issues and writing tests with ease, a new framework is introduced. Instead of writing code for the tests, the test information can be passed in a **property file**.

1. Create property file
2. Add test information
3. Pass file info in test suite
4. Run the test suite

The framework works as below

1. Parse property file
2. Create tests
3. Build request for each test
4. Send the request to the server
5. Receive the response and test it

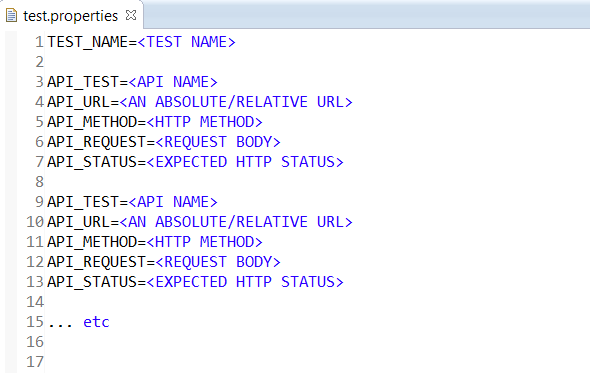
**RMA API testing framework**

# 5.1 Property files

Instead of writing code for the tests, pass the test information in a **.property** file. The framework converts these information into tests and runs it.

Here each property file acts as a test class. It contains test class name, and a list of tests. Each test from the list works as a network call.

The format of the file looks like below



Application’s API document and browser’s developer tools can be helpful in writing/creating property files.

# 5.2 Properties

In general, a property file contains data in **key-value** pairs. And they are separated by **‘=’**. Each line represents a different key-value pair.

Ex: PROPERTY\_NAME=PROPERTY\_VALUE

A test file can be created by adding below information.

* Test class block
* Test block list

There is a list of predefined **keys** which helps in passing test information. These keys are explained below.

**TEST CLASS BLOCK**

This block contains the information about the test class or the property file.

**1. TEST\_NAME**

As each property file works as a test class, this name is used while converting property file into test class. TEST\_NAME is useful in identifying errors and failures on runtime.

**TEST BLOCK**

A property file can have multiple test blocks. Each block acts as an individual test and represents a network (API) call.

A Test Block contains

* Request Information
* Test information

**1. API\_NAME**

It represents name of the test or API call. It helps in identifying successful/failure tests.

**2. API\_URL**

The server URL for the request. It can be relative or absolute. If the request has get/post params it should be added to the URL. (Appending using ?)

Ex: APIURL=/tweet/getTweets?page=1&limit=10

APIURL=https://www.api.twitter.com/tweet/create

**3. API\_METHOD**

An HTTP method of the request. It can be GET, POST, PUT or DELETE.

**4. API\_REQUEST**

Request body of the API Call. It is used for passing additional data in POST/PUT requests. It can be JSON, XML, Plain text or binary.

**5. API\_RESPONSE**

An expected response of the request. It can be a sample data format or a copy of real time response. It is useful while testing API call’s functionality.

**6. API\_STATUS**

An expected HTTP status of the request. This is used to evaluate the tests.

**7. TEST\_TYPE**

This field is to specify the type of testing method to compare expected response and actual response. It can be

* XML\_UNIT

Pass this, if the response is of type XML. It compares the data by comparing attributes and values of each element. Framework throws **assertion errors** if any of the values doesn't match.

* JSON\_UNIT

Pass this, if the response is of type JSON. It compares JSON data by comparing JSON Objects and JSON arrays and their properties. Framework throws **assertion errors** if any of the values doesn't match.

* STRING\_UNIT

Pass this to compare the exact strings.

* IGNORE\_VALUE

An XML or a JSON data can contain dynamic fields like IDs, Time Stamps, File paths ... etc. As these values won't match at runtime, framework throws an assertion.

To prevent these assertions, replace these dynamic values with the key word **IGNORE\_VALUE**. Framework ignores the unmatched values marked as **IGNORED** and prevents the assertion errors.

**8. VARIABLES**

A variable is useful in storing and using values at runtime. It's useful in handling dynamic fields in **URL**s or **Request body**.

**Ex:** If an API call response contains an **ID**, which can be used in future API call’s URL or Request. It can be stored as a variable and accessed when it is required.

The variable declaration should be passed as a **JSON/String** array.

**Ex:** ["VAR1=VARIABLE\_VALUE","VAR2=VARIABLE\_VALUE", ...]

Here **variable name** and **values** are separated by **'='**. A **VARIABLE\_VALUE** can be passed as

* JSON\_PATH

JSON\_PATH is useful in navigating through JSON data and obtaining specific value or object. Typically a JSONPATH starts with symbol **'$'** .

Ex:["book\_id=$.books[0].id"]

* XPATH

XPATH is useful in navigating through XML data and obtaining specific value or element. Use prefix **'XPATH:'** to differentiate it from other types.

Ex: ["book\_id=XPATH:/data/books[1]/@id"]

* CONSTANT

Any value other than of type XPATH and JSON\_PATH considered as a constant. Use **RANDOM\_VALUE** or **SYSTEM\_TIME** as part of the constant value to make it unique.

Ex: If a test is for creating a book entry with a unique name then, declare variable as **["book\_name=book\_RANDOM\_VALUE"]** and use it in **URL** or **request body**.

Variables can be used in **URL**s,**Requests** or **TEST\_CONDITIONs** by placing them in curly braces.

**Ex:**

API\_URL=/library/book/**{book\_id}**/details

API\_REQUEST={book\_name:"**{book\_name}"}**

Based on their scope, a variable can be declared under **SUITE\_VARIBALES** or **TEST\_VARIABLES**.

**8.1 SUITE\_VARIABLES**

Suite variable's scope is test suite level. These variables can be accessed in other tests across all the tests of the testsuite.

It's useful in resolving dependecies in API calls, which requires information from current API response.

**8.2 TEST\_VARIABLES**

Test variables scope is reduced to individual test, and used for temporary purpose. These variables can be used in request body, parameter values or test conditions.

**9. TEST\_CONDITION**

Testing API call based on HTTP status doesn't assure the complete testing of the functionality. And comparing requests using **TEST\_TYPE** is heavy, as it compares whole response data.

To test only a specific data, a test condition can be written using variables. This condition similar to if condition of java or JavaScript. The test result depends on the execution result of the condition.

Ex: **TEST\_CONDITION**=(({books\_available}==2) && ({last\_book\_name}=='My Book'))

**10. API\_POLL**

In few scenarios, application polls API requests until they get a particular response. To handle these kind of scenarios, mark the request as **'POLL'** by passing

**API\_POLL=true**

The framework polls the request multiple times until the test passes or poll time expires. The default poll time is **5 minutes** and interval is **10 seconds**.

To override these values use **POLL\_INTERVAL** and **POLL\_TIME** and pass the time in seconds. **API\_STATUS**, **TEST\_TYPE** and **TEST\_CONDITION** are crucial in proper polling of a request.

**11. WAIT\_TIME**

To force the API call to wait for some time before contacting server, use this property and pass the time in seconds.

These property files should be stored under project’s **src/test/resources/<test folder name>.**

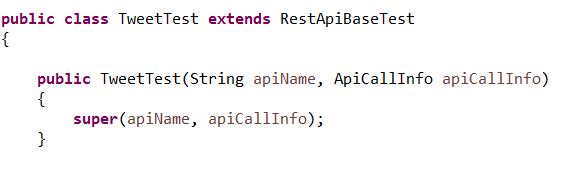
# 5.4 Test Class

Each **property** file represents a test class. To run the property file as a test class, it need to be passed from the test suite. Based on the test information of the property file, the framework converts the property file into a test class.

If the requirement is to customize the test class and monitor the API calls of the property file, it can be done by creating a custom test class.

1. Create test class:

Create a child class by extending **RestApiBaseTest.java**



This constroctor is called for each test/API call of the property file. **apiCallInfo** passed inside the constorctor contains all the test/API call information.

1. **Configure test:**

Pass the property file path, by creating a mandatory method with below signature

@Parameters(name = "{0}")

**public** **static** Collection<Object[]> data() **throws** Exception{

*setPropertyFile*("Twitter/tweet\_test.properties");

**return** RestApiBaseTest.*data*();

}

1. **Lifecycle:**

Test class’s lifecycle can be tracked by overriding below super class methods.

* setUp

This method gets called before running the test class. It’s used for initialization.

* tearDown

This method gets called after running all the tests of the property file. It’s helpful in clearing setup data.

* handleApiRequest

This method gets called before running each test block. As each test block represents an API call, this method is useful in customizing API requests.

* handleApiResponse

This method gets called after sending API call to the server. It is useful in monitoring API response data and customize testing.

# Test Suite

A test suite is useful in initiating test process. The property files and test classes can be grouped together in a suite.

Create a java class by extending **RestApiTestSuite** class. Add below mandatory method, and pass property file and tests information.

**public** **static** TestSuite suite() {

// creating a list of property files and test classes to be tested

List<Object> testList = **new** ArrayList<>();

testList.add("component\_folder/property\_file1.properties");

testList.add(CustomPropertyTest.**class**);

testList.add("component\_folder/property\_file2.properties");

testList.add("component\_folder/property\_file3.properties");

// Passing current package path and test list

*init*(SampleTestSuite.**class**, testList);

**return** RestApiTestSuite.*suite*();

}

These tests and test suites should be kept under ***src/test/java/<Component Name>***

Override below methods to track lifecycle of suite and test classes.

* setUpSuite

This method gets called before running tests. It is useful in setting up test suite configuration.

* tearDownSuite

This method gets called after finishing all the tests. It useful in clearing the configuration settings.

* setUpClass

This method gets called before running each test class/property file. **ApiTestInfo** is passed as parameter and it holds the test information.

* tearDownClass

This method gets called after running each test classes/property files. **ApiTestInfo** is passed as parameter and it holds the test information.

A test suite can be configured as **JUNIT** type, and started running from eclipse/Jenkins.

# 6 Test Configuration

Test suite configuration is useful in changing the test behavior at runtime. It helps in passing test information and configuration settings.

* Base URL

Typically test information inside property file contains relative URL. Base URL of the server/application helps in creating full/absolute URL at runtime.

* Authenticator

To prevent unauthorized usage, users need to be authorized and get access before using any modern applications. Applications provides access-tokens which can be used while sending requests to the server. Typically these access-token information is passed as HTTP headers. Omit this, if the application is read-only and doesn’t require any authentication or authorization.

It works as below in the framework.

1. Create an authenticator class by extending RestAuthenticator.
2. Override setUp method.
3. Fetch access token details from the application for the test user.
4. Add these details to the HTTP headers by using test suite’s RestNetworkUtil
5. All the API calls inside test classes uses these headers to get the access.

* Performance Tracker

Performance tracker helps in recording the time taken by each API call. These records are helpful in identifying sudden performance changes of the server.

* Record Location

By default, all the recorded performances are stored as .csv files under **src/test/resources/performance** folder. Use this setting to pass the custom location.

* **Append Results**

Use this setting to merge the records in continuous runs.

Sample code:

RestSuiteConfiguration config =

**new** RestSuiteConfiguration.Builder()

.setBaseUrl(<Base URL>)

.setRestAuthenticator(<app authenticator>)

.setTrackPerformance(<true | false>)

.setAppendPerformanceResults(<true | false>)

.setPerformanceRecordLocation(<location>)

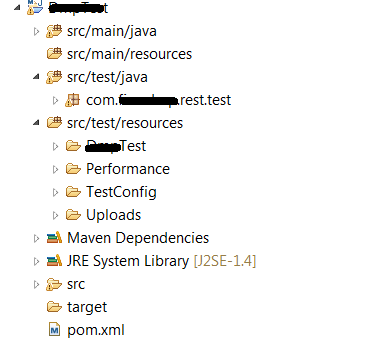
.build();

initConfig(config);

This configuration should be passed from test suite’s **setUp** method.

# 7 Project Setup and Flow

1. Create a new **java-maven** project in eclipse.



2. Download the library and import it as maven project. Add the library as a dependency project.

3. Create and add property files (tests) inside **src/test/resources**

4. Add helper classes insde **src/main/java**

5. Add test suites and custom test classes under **src/test/java**

6. Add configuration settings inside test suite. To make the settings more dynamic, pass the test configuration as system variables.

7. Create a JUNIT test configuration and run the test suite.

# 8 File Upload requests

Uploading files to the server is little different from normal APIs. Pass the file information as part of the request body.

1. Place the upload files under resource/uploads folder.
2. Create a request using below format.

REQUEST=**UPLOAD\_REQUEST;<file path>;<request-payload>**

Ex:

REQUEST=UPLOAD\_REQUEST;uploads/profile\_data.csv;uploadFile|formdata1=data|formdata2=data

Here:

**UPLOAD\_REQUEST :** Keyword to help the framework in identifying the request.

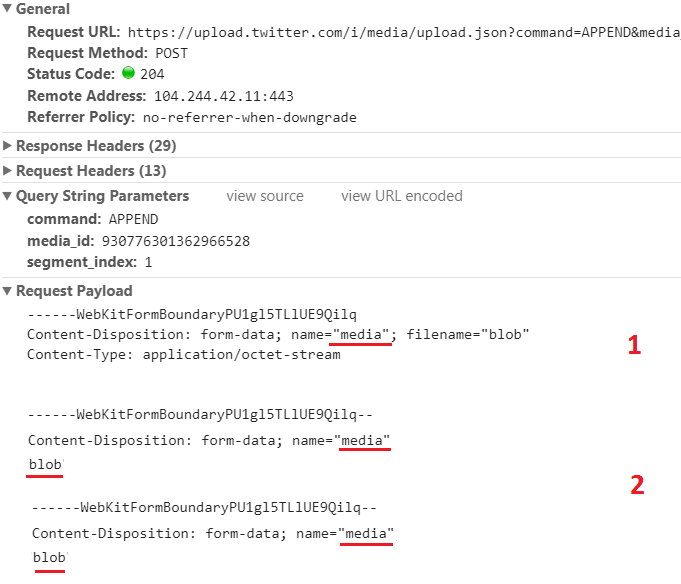
**File path :** Paths of the files to be uploaded (separated by comma).

**Request payload (Optional):**

Content-Disposition name from the request payload and all the form data names and values separated by pipe (|).

3. All these fields should be separated by semi colon (;)

Please refer below screen shot take from chrome developer tools.



* 1. Form data name (From first Content-Disposition)
  2. Form data params

# 9 File Download requests

Downloading files from server is similar to **UPLOAD**.

1. Create a request using below format

REQUEST=**DOWNLOAD\_REQUEST**

Ex:

REQUEST=DOWNLOAD\_REQUEST

Here:

**DOWNLOAD\_REQUEST** : Keyword to help the framework in identifying the request

The file will be downloaded to **src/test/resource/downloads** folder.

# 10 Chrome Extension

Creating test property files are time consuming and requires good knowledge of browser developer tools and API document (swagger etc).

Below are the common issues one can face while writing tests.

1. Identifying API calls from developer console.
2. Manually copying requests and responses
3. Creating property file and adding tests with the specific format.
4. Monitoring API calls on the fly
5. Ignoring unnecessary calls
6. Writing test cases in parallel while testing defect/feature

# 11 Resources

Json Unit - <https://github.com/skyscreamer/JSONassert>

XML Unit - <https://github.com/xmlunit/xmlunit>

XPath - <https://en.wikipedia.org/wiki/XPath>

Json Path - <http://goessner.net/articles/JsonPath/index.html>

Chrome Extension - https://github.com/srinivas2207/RestApiChromeExtension