# **Performance Testing of Platzi Fake Store API (RESTful API).**

-*Nikesh Man Shrestha*

For this project, I used a demo website known as ‘Platzi Fake Store API’, a public RESTful e-commerce API – perfect for load & stress testing providing a variety of endpoints to test.

**URL**: https://fakeapi.platzi.com/ (Project Demo site)

## **Step 1: Defining the project goal**

*“Simulating real-word traffic on demo e-commerce website API to measure the system’s performance under normal and high user load.*

## Test Scenarios:

1. Load test fetching product list: GET/ product
2. Load test product details: GET/ products{id}
3. Load test login: POST/auth/login (using test credentials)
4. Load test add to cart: POST/ cart
5. Stress test user search: GET/ users

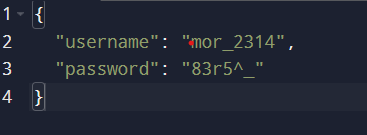
## **Step 2: Identification of key Test cases**

Let’s break down the specific REST API request to simulate in Jmeter.

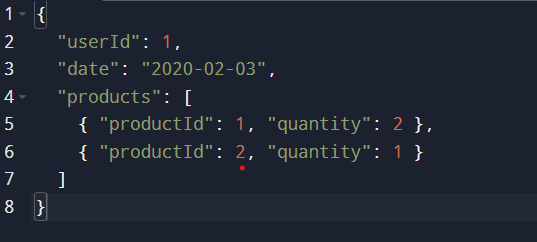
Here are the detailed test cases I considered for the following project.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Scenario Description | Endpoint | Method |
| TC01 | View all products | /api/v1/products | GET |
| TC02 | View single product detail | /api/v1/products/${product\_id} | GET |
| TC03 | User login with test credentials | /auth/login | POST |
| TC04 | Add products to the cart (basic payload) [NOT TESTABLE] | /carts | POST |
| TC05 | Get list of users | /users | GET |
| TC06 | Get product by id (Parametrized test through csv file) | /api/v1/products/${id} | GET |
| TC07 | Get all categories | /api/v1/categories | GET |
| TC08 | Create a product | /api/v1/products | POST |
| TC09 | Update a product | /api/v1/products | PUT |
| TC10 | Delete a product | /api/v1/products/product\_id | DELETE |

So, a sample payload for Login (TC03) would be



Similarly, sample payload for Add to cart(TC04) would be:



## **Step 3: Defining the functional flow of requests**

For simulating the requests, let us take the realistic user journey. For that, functional flow would be something like this.

1. Login- Authenticate using valid username & password.
2. Browse products- View all available products.
3. Browse categories- View all categories.
4. View details- Click to view a single product’s detail.
5. Add to cart- Simulate adding selected product to cart. [NOT TESTABLE IN API]
6. Fetch users- Load user data for internal processing/reporting.

This flow ensures you simulate both **read-heavy** (GET) and **write-heavy** (POST) operations, which is crucial for performance testing.

Execution Plan (for Jmeter Setup):

TC03: POST /auth/login

TC01: GET /products

TC02: GET /products/1 (Can be parametrized later)

TC04: POST /carts

TC05: GET /users

TC06: GET/ products by id

TC07: POST/ create product

TC08: PUT/ update product

TC09: DELETE/ delete product

## **Step 4: Create JMeter Test Plan & Thread Group Setup**

**Creating a Test Plan and Thread Group setup:** Initially, a Thread Group is created and initialized with the following test settings:

|  |  |  |
| --- | --- | --- |
| Field | Value | Notes |
| Number of Threads | 10 | Simulates 10 virtual users |
| Ramp-Up Period | 10 | 1 user starts every second |
| Loop Count | 1 | Each user runs 1 cycle |

These settings are later adjusted for load/stress testing

## **Case 1.1: Adding Login Request in the Thread Group (TC03)**

1. **Adding HTTP Request Sampler for Login API:** A HTTP Request Sampler is added to simulate POST/ auth/login request. The steps covered for this:
   1. Adding HTTP Request
   2. Configuring the Login Request

|  |  |
| --- | --- |
| Field | Value |
| Name | Login – POST /auth/login |
| Method | POST |
| Server Name | api.escuelajs.co |
| Path | /api/v1/auth/login |

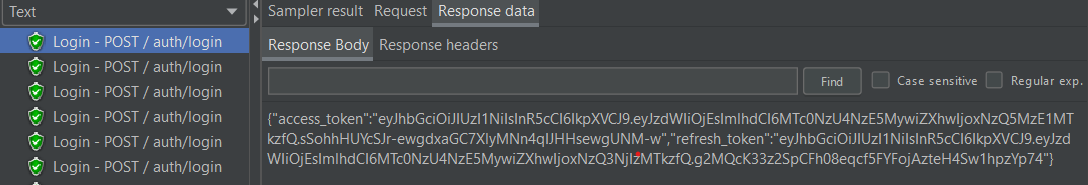
Body



* 1. Setting the Header for Content-Type:
     1. Adding the Header Manager: Right Click on Sampler =>Add-> Config Element -> HTTP Header Manager
     2. Click Add; Name: Content-Type, Value: application-json
  2. Adding Listener to View Result: View Results Tree Listener [Added]

1. **Adding Json Extractor Post-processor**: Json extractor Post-Processor should be used when you are testing an API that returns dynamic values (in our case, access token) that needs to be reused for future requests. The Extractor is configured as:

|  |  |
| --- | --- |
| Field | Value |
| Name | Extract Access Token |
| Variable Name | access\_token |
| JSON Path Expressions | $.access\_token |
| Match No. | 1 |
| Default Value | NOT\_FOUND |

1. **For using the extracted token in Subsequent Requests**: An HTTP Header Manager can be added with Header Name: Authorization, Header Value: Bearer ${access\_token}
2. **Run the Login Sampler:**  To check whether the Login request works or not, The JSON response looks something like this:  
   

## **Case 1.2: Add GET/ products Request (with Auth Header)**

*Simulating the next user action i.e. fetching a list of products after login*

1. **Add the HTTP Request Sampler for GET products:** Same as Adding sampler for Login but method is GET
2. **Add Authorization Header**

*Upon running the test, the following observations is seen.*

1. Status code is 200
2. Response contains the list of products in JSON format.

## **Case 1.3: Add GET/ products by id with id being dynamically supplied**

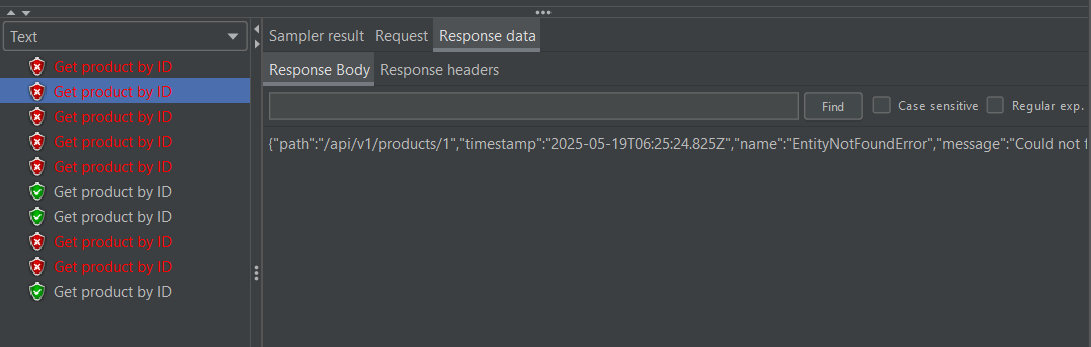
*Simulating multiple get request for product using parametrized id through csv file*

1. **Add a csv file to the project folder:** Single column csv ‘products\_id.csv’ is created with column title ‘id’ and values of id in each row.
2. **Add the HTTP Request Sampler for GET product by ID:** Same as others; *path:/api/v1/products/${id}*
3. **Add CSV Data Set Config Element:** The config element is initialized as

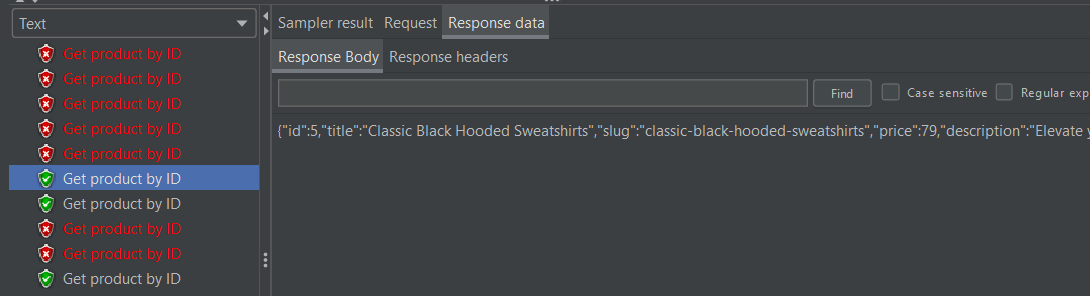
|  |  |
| --- | --- |
| Field | Value |
| Filename | products\_id.csv |
| Variable Name | id |
| Delimiter | , |
| Recycle on EOF? | True |
| Stop thread on EOF? | False |
| Sharing mode | All Threads |

***Upon running the requests, observations seen:***

* The ids in csv were given from 1-15, the deleted items in this range showed status code ‘400’ Bad Request with error response data.



* For the items present in the range, status code ‘200’ OK with item details in the response data.



## **Case 1.4** **Add GET/ products/{id} to view an Item**

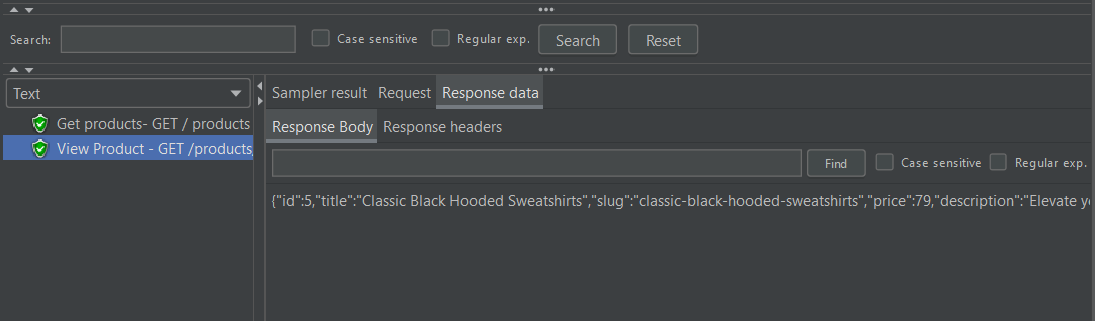
*Viewing the first product in the list obtained from the GET/ products Request*

1. **Add Post-Processor JSON Extractor:** To extract the product id from the list of products fetched from the GET/ products request in Step 5.2. The extractor is initialized as:

|  |  |
| --- | --- |
| Field | Value |
| Name | Extract Product ID |
| Variable Name | product\_id |
| JSON Path Expression | $[0].id *(note*: *this gets the ID of first product in the list*) |
| Match No. | 1 |
| Default | NOT\_FOUND |

1. **Add the HTTP Request Sampler for View products:**  Same process
2. **Add Authorization Header:** With Header; Authorization having value: ‘Bearer ${access\_token}

For 1 user, the request looks something like this

:

## **Case 1.5: Add GET all categories request**

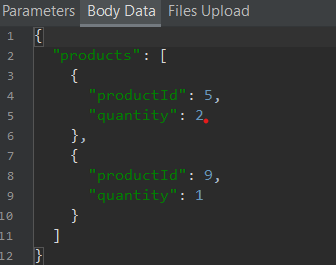
1. Add new request for GET categories
2. Adding a Constant Timer: Setting the Thread Delay of 1000ms (1 second between each request)
3. This request executed under usual thread configuration of (10 threads, loop count 5, ramp up 5)

*Upon observation, the response returned ‘Status 200 OK’ within reasonable load time (<1s for Get response).*

## **Case 1.6: Add Cart Creation POST request [NOT TESTABLE]**

*Goal: To simulate multiple add to cart transactions – to observe api performance under load.*

1. Add a POST HTTP request for cart creation
2. Adding a JSON Payload in the HTTP request.



1. Add Header with name: ‘Content-Type’, Value: ‘application/json’
2. Test ran with a few users (i.e. 5 threads(users), loop 2)

*Upon completion of requests,*

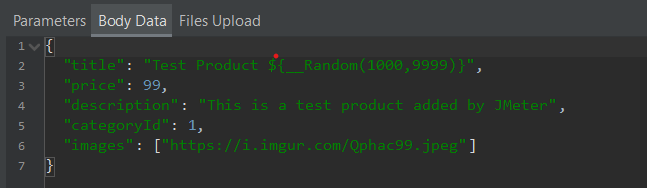
HTTP status code: 201 Created, response body should return a cart ID or a list of products.

## **Case 1.7: Load Testing Create product request**

*Goal: To simulate multiple users creating products – each with slightly different title — to stress test the API.*

For this I created a new Thread Group, and initialized Number of threads(users):10, Ramp-Up Period: 5, Loop Count: 5. This means 10 users will send requests repeatedly for 5 times [Total 50 requests]

To avoid the duplicity of product creation, I used a Jmeter’s built-in function and sent the body data in the following format.



**Observations:**

* 50 sample requests were sent with Avg response time: 652ms, Error %: 0%
* Min response time: 376ms and Max response time: 1417ms

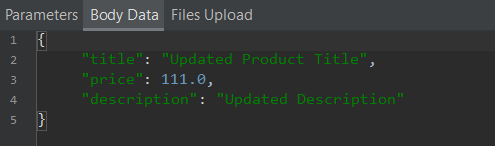
**Assertions:**

Furthermore, In the end stages of the project, I included assertions for status code and whether the response data contains ‘id’, ‘title’ and ‘description’. All the values in the response data can be added.

## **Case 1.8: Add Load Testing Update product request**

For this, I added Two HTTP requests; one for load testing the Update product requests and another for single product update request and its assertion.

So, for single product update request, the body data is sent as:



Assertions are made according to this body data and checked.

**Observations**:

* Response code is ‘200 OK’
* Average response time is 951ms for 10 sample requests.
* ***Error found:*** *When the price is updated in float, the API discards the value after decimal and converts the given float price into ‘int’ which might cause issues in the application.*

## **Case 1.9: Delete request testing and assertion**

The demo API used for the project returns HTTP status ‘200 OK’ when an object is deleted with response data true. For this status code response assertion is added under the Delete request HTTP.

## **Case 2: Performing Endurance Test – Continuous load for 5 minutes**

1. Initial Setup: Thread Group is set up as:
   1. Number of Threads: 10 (initially)
   2. Ramp-Up Period: 10
   3. Loop Count: changed to Infinite
2. Selection of two requests: GET all products and GET all categories.
3. Add Summary Report Listener and View Results Tree.
4. Duration Assertion is added to monitor response time, failing the tests that require more than a threshold value(>2500ms).

*Upon running of the tests, the following observations was done:*

**Observations**:

* All responses return HTTP status “200 OK”
* Reasonable response time under load
  + Get all products, Avg: 637ms (*<1000ms is great for GET requests*)
  + Get all categories, Avg: 257ms
  + Average Error%= 0.22% ~ 0% (*error occurred during the manual stop of the requests after 5 mins and not while making the requests*)

