***Documentation for JMeter Load Testing***

JMeter is a free application, used for load testing an application or a web site, so it can benchmark the limits while there is a big traffic at a time visiting/using the app or website. While load testing is performed, we simulate loads of traffic and typical user activities on the app/website.

There are 2 things the tester gets as feedback:

1. How the system works under loads of visitors/users
2. The number of resources used and needed

How do you set up the app, to get it ready for testing:

1. On the device you will execute the tests, you need to download Java Virtual Machine ([JVM download link](https://www.java.com/en/download/))

A screenshot of a computer

AI-generated content may be incorrect.

1. Then you download the JMeter application from the following link ([JMeter](https://jmeter.apache.org/download_jmeter.cgi))

A screenshot of a computer

AI-generated content may be incorrect.

1. After downloading the .zip file, you need to extract it first, then go to bin and we look for jmeter.bat. If the extension is not written, you can do it by clicking right click to the files and check the extension in their properties.

A screenshot of a computer screen

AI-generated content may be incorrect.

1. After the file is found, double click it and you are going to see a terminal in one window, and in another window the app GUI.

A screenshot of a computer

AI-generated content may be incorrect.

1. A screenshot of a computer

   AI-generated content may be incorrect.A screenshot of a computer

   AI-generated content may be incorrect.Now the application is runed, and the testing can start preparing. The first thing after the app is runed, need to be the ‘Test Plan Building’
2. Now the test plan is created, named with a descriptive name and the scenarios writing can start being written.

A screenshot of a computer

AI-generated content may be incorrect.

Test scenarios describe specific actions or use cases that need to be simulated, so it can give feedback about the system load performance. They help understanding how the application behaves under certain conditions.

In the context of JMeter, a test scenario includes details like the number of virtual users, the type of HTTP request (GET, POST, etc.), the target URL or endpoint, data input, and expected results. Scenarios are created using Thread Groups, HTTP Samplers, etc....

**Scenario1: Load testing a /get request**

A screenshot of a computer

AI-generated content may be incorrect.

This Thread Group will be used for the next //TO\_FIX number of scenarios.

Explanation of the setup:

So, JMeter has 50 virtual users (Number of Threads), then 50 users, over 10 seconds (Ramp-Up), it means that 50/10 -> the test will start 5 users per second. And there is no infinite loop, besides there is 1 request per user, not more.

A screenshot of a computer

AI-generated content may be incorrect.

Explanation of the test result:

Server handled 50 /get requests with **no errors**. The **average response time is 676ms**, which is decent. The response time range is [**231,1775] ms**, meaning that some spikes occurred, likely due to simulated concurrency. The **throughput** of 4.7/sec is normal given your ramp-up of 10 seconds for 50 users.

*This is a basic load test to measure server performance under light concurrent traffic.*

**Scenario2: /post request with JSON body**

This scenario, we create a post request, with a body of a JSON string. A screenshot of a computer

AI-generated content may be incorrect.

Then we add Http Header, because we need to specify the content we we are putting in the body.

A black and grey striped background

AI-generated content may be incorrect.

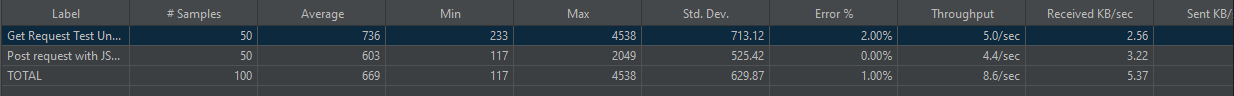
We add a tree result, where we can see for each thread the request it is making, how the response is.

A screenshot of a computer program

AI-generated content may be incorrect.

Now the testing is started again, under the same thread group configuration as before.

Here there are the results:



Explanation of the test result:

The POST request with the provided JSON string in the body sent **50 post requests** with to the /post endpoint of httpbin.org. The goal was to check if the server received and echoed the JSON correctly, and to observe performance under a light load. The average response time was 603ms, which was slightly faster than the previous GET request test This test validated the functional correctness and stability of POST request handling under light load.

**Scenario3: simulating slow API response**

The third scenario is made to test slow API response, using a delay of 5 seconds for a /get request. We set the test to <https://httpbin.org/delay/5>, and we need to check if there is going to be a delay of 5 seconds, and then the response.

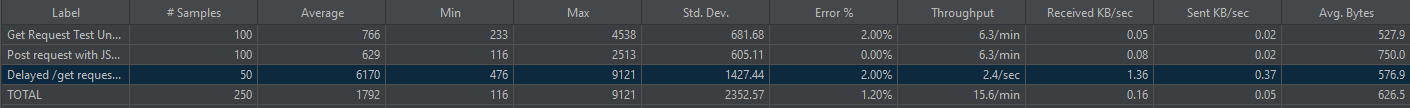
Simply, it allows us to evaluate how well our system can tolerate such latency and still function correctly.

Here is the setup:

A screenshot of a computer

AI-generated content may be incorrect.

Here is the result:



Explanation of the test result:

We sent 50 requests with a moderate number of users. The average response time was 6170 ms, which aligns with the expected delay. The system experienced 2% of failed requests, which could be caused by timeouts or temporary overload. The test helps simulate real-world latency and observe how the system performs under delayed API responses. The throughput dropped to 2.4/sec, demonstrating that slow endpoints reduce overall performance.

**Scenario4: Basic authentication test**

This scenario is meant to see if the login is successful/unsuccessful. It means that, if the login is successful, it will return status code 200 OK, otherwise 401 Unauthorized.

We need to set a new request, where we set the path.

A screenshot of a computer

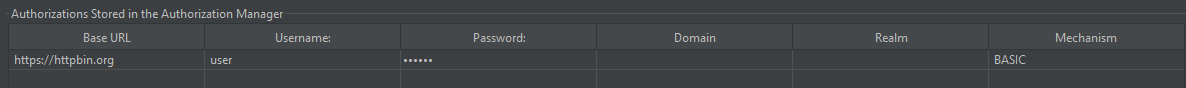
AI-generated content may be incorrect.

We set a Http Request Defaults, so it can use the https protocol, for more safety, and I set the domain httpbin.org, so it can be the same for all requests.

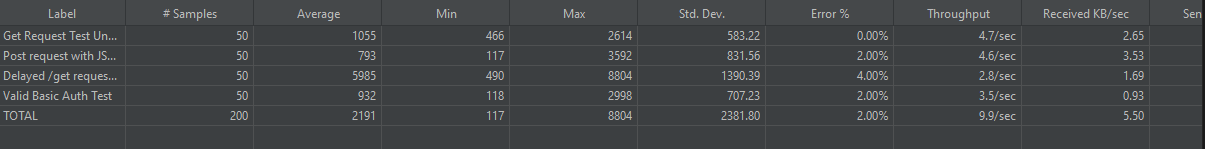
Then we set a new file, called Http Authentication Manager, where we set the password and the username that need to be provided to the searched domain/path, so it can be logged in.

A screenshot of a computer

AI-generated content may be incorrect.



And after testing, we got the following results:



Explanation of the test result:

In this scenario, we tested HTTP Basic Authentication by sending a GET request to /basic-auth/user/passwd with valid credentials. We used JMeter’s HTTP Authorization Manager to provide the username and password. Out of 50 requests, 49 completed successfully (98%), and 1 request failed (2%) *likely due to temporary connection issues*. The test successfully demonstrated correct server-side authentication handling.

**Scenario5: Testing different http status codes**

Here in this scenario, we made testing on some status codes. First, and most important thing, we create the http requests for all of them separate. We choose to test 200, 404, 500 status codes. Then on the request we create an assertion that is going to make a test check if the test code we got, as the one we defined (the code we expect).

Here we will show just one request and assertion, for example let it be **404 Not Found** (the others are the same, but with different code)

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Note: For the test to be evaluated as true or false, we need to check Ignore Status in the assertion, otherwise we will always get error when we set error codes (the point of the scenario is to check if the status can be accessed).

A screenshot of a computer

AI-generated content may be incorrect.

Explanation of the test result:

This scenario confirms that the application consistently handles various HTTP response codes as expected, which is essential for robust error handling, client-side logic, and backend monitoring. The minor error percentages observed may be attributed to differences between the expected and actual status codes, caused by transient server or client-side issues during high-load conditions.

**Scenario6: Cookie handling test**

This test is going to check if a set cookie, can be recognized and retrieved.

We create 3 files now:

A grey background with white text

AI-generated content may be incorrect.

The cookie manager allows storing and sending cookies (the communication between them)

A screenshot of a computer

AI-generated content may be incorrect.

Then we got the request with cookie, here we are creating a simple get request where we set?sessionId=skit\_project\_cookie123, so it is a trigger to the server for setting a cookie.

A black and white striped background

AI-generated content may be incorrect.

And the cookie retriever is a request that is trying to retrieve the cookies that are already set. So in the first request we set, and here we see if the cookie exists. Additionally we set an assertion to the retriever request, where we see if the cookie is set correctly.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.On the screenshot below, we can see that the body of the request is giving that the cookie exists and gives its sessionId.

*Request with cookie-0 - (/cookies/set)* ***sets*** *the cookie and then* ***redirects*** *to /cookies.*

*Request with cookie-1 - (/cookies) is* ***explicitly added*** *to verify the cookie value.*

A screenshot of a computer

AI-generated content may be incorrect.

Explanation of the test result:

The test confirms successful cookie management between requests. The redirect in the first request increases the average response time and accounts for some variability. The second request validates that the cookie is correctly stored and reused, with stable performance and zero errors. This test is essential for validating session persistence and client tracking, effectively introducing stateful behavior into an otherwise stateless HTTP protocol.

**Load Test with 10 users at a time:**

Then we create a new Thread Manager, with 10 users at a time, all of them will have access on a period of 5/10 seconds and will have to access it 2 times.

**Scenario 1 – Http Put**

Here we test the method put, where on /put we insert a json that updates update=true. We check it by adding response body assertion that checks if the json is contained in the updated object.

**Scenario 2 – Redirect Request**

Here we want to achieve redirecting to another page /redirect-to?url=https://httpbin.org/get. And we check if the page url is the same with the url provided for redirection.

**Scenario 3 – Ip Detection Request**

Here we try to detect if on /ip, it will return an ip address.

**Scenario 4 – Check Host Header**

Here it checks the host header mapped on /headers.

**Scenario 5 – Base64 – Echo**

It goes to an endpoint /base64/encoded string}, where the encoded string is decoded and in the assertion, we check if the meaning of the encoded string is the one we like to be returned.

**Load Testing Fake Store Application (ReqRes app)**

Description of the testing:

This thread manager is going to test a fake application that is created for api testing, and I will load test it ([app\_link](https://fakestoreapi.com/)). I will use 100 users, every of them will start on a period of 10/100 seconds.

**Scenario 1 - Get Users Page = 2**

This is going to fetch all the users from the second page and check if there is a user and to get his email. If this is done, it means that there is a list of users. We will set to check from the data[1], so it can check if there is a list returned. This will check if users are successfully fetched.

**Scenario 2 – Get just one user (single user)**

This will load tests by fetching a single user (in our case with id=2). The check is going to be made with Json assertion, to check if the fetched object has id, that means that there is a data (user) returned.

**Scenario 3 – Get non-existing user**

This scenario validates the system properly handles cases where a user does not exist. When querying /api/users/404, the server returned a 404 response code, which was verified using a Response Assertion. This is important for verifying error handling and client feedback mechanisms under heavy load.

**Scenario 4 – Creating user using POST method**

In this scenario, we simulated creating a new user using the /api/users POST endpoint. We validated that the server returned a 201 Created status and that the response JSON by checking if there is a generated ID inside, and a generated ID.

**Scenario 5 – Update Existing User, with id=2**

Here we are updating the user with id=2, by putting new name and job to it. We make the check with json assertion, where we set the new expected job label to be updated.

**Scenario 6 - Delete User**

Here we test deleting the user with id = 2. The application simulates deleting the same user, that is going to prove the concept of ‘what if many users delete at same time, will it perform good’. We check if the code 204 for successful delete is returned.

**Scenario 7 - Delayed response handling**

We check here the delay of 3 seconds, if the delay expires and the response code is 200, it means that the get method is successful and the test passed. Also we added a check, for the duration not to be over 4200 milliseconds.

**Scenario 8 - New user registration**

Here we tested successful user registration via /api/register by submitting valid credentials. The server responded with HTTP 200 and returned a JSON token. Assertions ensured both the token field and response code were valid.

**Scenario 9 – Registering fails (missing password)**

Opposite of scenario 8, this is a negative test case that checks how the API handles incomplete user input during registration. When the password is missing, the /api/register endpoint returns a 400 Bad Request with an appropriate error message. We catch the error by checking if the code is 400 and the register is failed with error message.

**Scenario 10 - Login Successfully Done**

This scenario tested the login functionality via the /api/login endpoint using valid credentials. The server responded with a 200 OK and returned a valid authentication token. Assertions verified the presence of the token field and correct status code, ensuring the login flow is functioning reliably under simulated user load.

Now we will provide some images of the load testing with different Thead Manager Configurations

1. Number of Threads (users): 65

Ramp-up Period (seconds): 5

Loop Count: 1

A screenshot of a computer

AI-generated content may be incorrect.

Catastrophically output, because the load of 65 users, each user differentiated by a period of 5/65 seconds with one try each is not going well. It gives so bad testing results

1. Number of Threads (users): 65

Ramp-up Period (seconds): 15

Loop Count: 1

A screen shot of a computer

AI-generated content may be incorrect.

Now the output is a bit better, still not as good as we need. Especially, when we log in, when we register, the most critical parts are shown as bad while big traffic visit .

A screenshot of a computer program

AI-generated content may be incorrect.

All the errors are Too Many Requests

1. Number of Threads (users): 30

Ramp-up Period (seconds): 15

Loop Count: 1

A screenshot of a computer

AI-generated content may be incorrect.

Slightly better under less traffic.

1. The results when we have 1 user only are good, but it does mean that the application is meant to be used by only 1 or 2 users at a time. Otherwise, it will have bad results, because the server will be overheated with so many requests and only 429 – Too Many Requests will show.

A screenshot of a computer

AI-generated content may be incorrect.

Conclusion:

This project demonstrates the power and flexibility of Apache JMeter for both load and functional testing. Through multiple test scenarios using public APIs like [httpbin.org](https://httpbin.org) and [reqres.in](https://reqres.in), we were able to validate:

* System behavior under concurrent user load.
* Correct handling of different HTTP methods (GET, POST, PUT, DELETE).
* Response verification through assertions (status codes, JSON responses, redirects, cookies, headers).

The ReqRes application specifically revealed limitations under high load (many 429 “Too Many Requests” errors), highlighting its unsuitability for high-traffic environments without proper throttling or load balancing.

By creating structured thread groups, adding assertions, and analyzing metrics like response time, throughput, and error percentage, we demonstrated how JMeter can be used not just for automation but as a powerful tool for **simulating production-like traffic**, improving **reliability**, **performance**, and **scalability** of web APIs and applications.

JMeter, when used correctly, is an essential asset in a QA or DevOps pipeline for identifying bottlenecks and ensuring readiness of the application for the real world usage.