Load Testing Using Apache JMeter

Mitesh
Jan 1 · 6 min read

Photo by VanveenJF on Unsplash

Load testing is the process of putting the load through (HTTP, HTTPS, WebSocket etc) calls on any software system to determine its behavior under normal and high load conditions. Load test helps identify maximum requests a software system can handle. It helps determine the point of a bottleneck due to which application starts degrading performance. Load testing is effective when we simulate real user scenario. We need to know how our users behave on our application. Try to replicate their behavior using different API calls and generate load through many concurrent requests at our application for an extended period of time to understand application behavior changes.

When we generate load on the system beyond normal usage (high load) to understand application/system behavior, we call it stress testing. Stress testing is done to know the limits of an overall system, understand kind of errors thrown during such load and how the end user is affected. This helps in planning for events when a much higher load is expected by provisioning more infrastructure or optimizing applications.

We can use Apache JMeter for load testing static as well as dynamic resources which can generate concurrent users to simulate real user scenario. It also provides graphical analyses of test reports. It is written in Java, so we need to have java runtime installed to run Jmeter. As per JMeter website, it is defined as mentioned below :

> *The Apache JMeter™ application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.*

Install JMeter on your system from where you want to run your test. I am using Mac OS for running Jmeter and using brew to install JMeter.

```
brew install jmeter
```

We first need to check file descriptors we can open on our system using below command. Every OS has limits on active file descriptors. Every TCP connection takes a file descriptor, so we need to make sure we have enough file descriptor (can set between 1–65535) to run concurrent requests.

```
ulimit -n
```

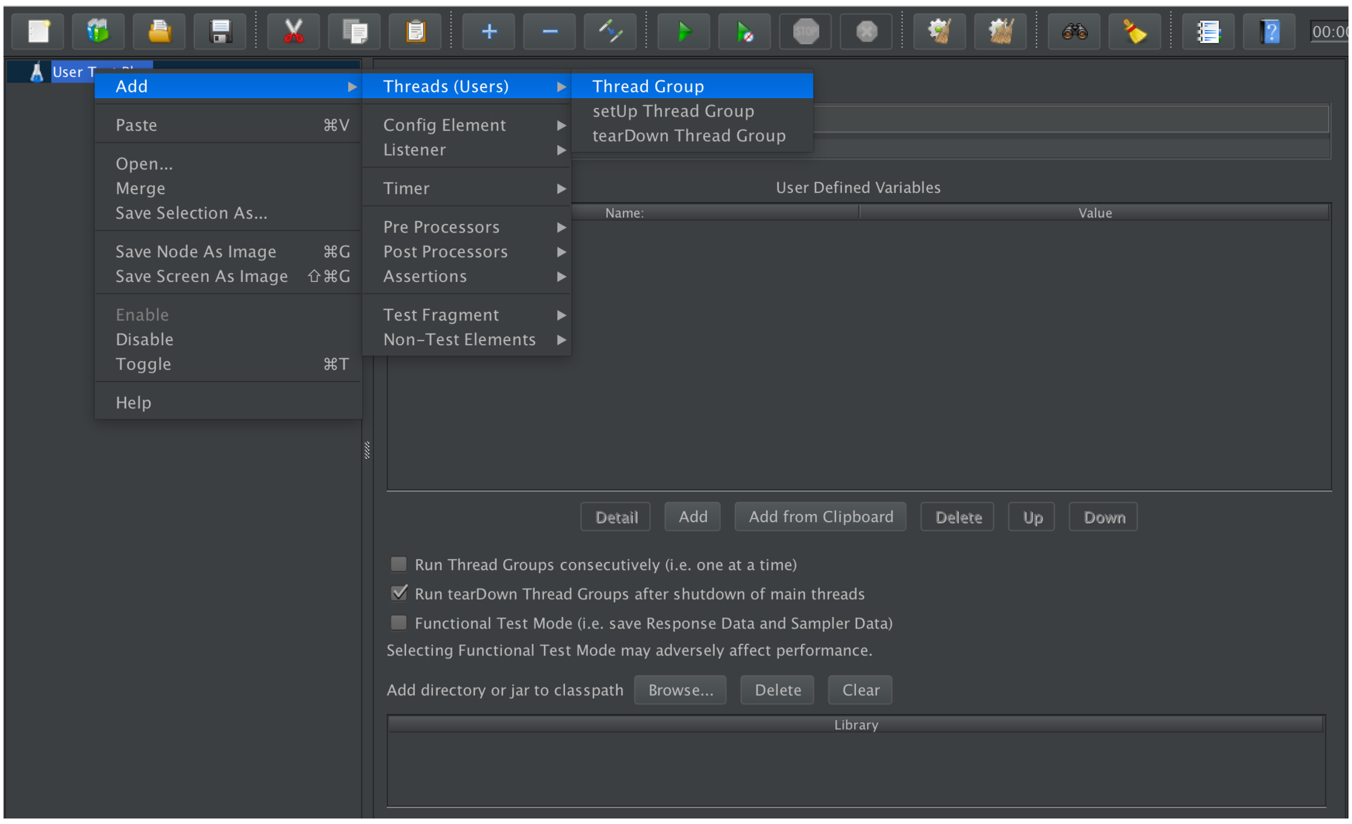
To start our load test, we need to start with thread groups. They are the beginning point of any test plan. We can specify the number of users (= number of threads) that we want to simulate concurrently. Set ramp-up time which specifies the amount of time it takes to start all threads. We need to define a number of iterations for each user(thread) in the group. Each thread executes the plan independently from other threads. All elements inside single thread will execute sequentially.

We can add samplers and logic controllers in a thread group. Samplers help send a request to a server and wait for response i.e. HTTP request sampler to send HTTP request and get the response. Logic controllers help customize the logic to decide when to send a request, modify request etc.

Listeners are another type of element which we can add in a thread group. They provide access to information about our load test. Graph result listener helps plot results of our test. View result tree listener shows the detail of request and response. We can add pre-processor and post-processor to a sampler request. Pre-processor executes defined action before sample request and post-processor executes defined action after sample request. Pre-processor can be used to generate dynamic parameters different for each request and post-processor to fetch response parameters to be used by other requests. Config elements are used to configure default value for the samplers.

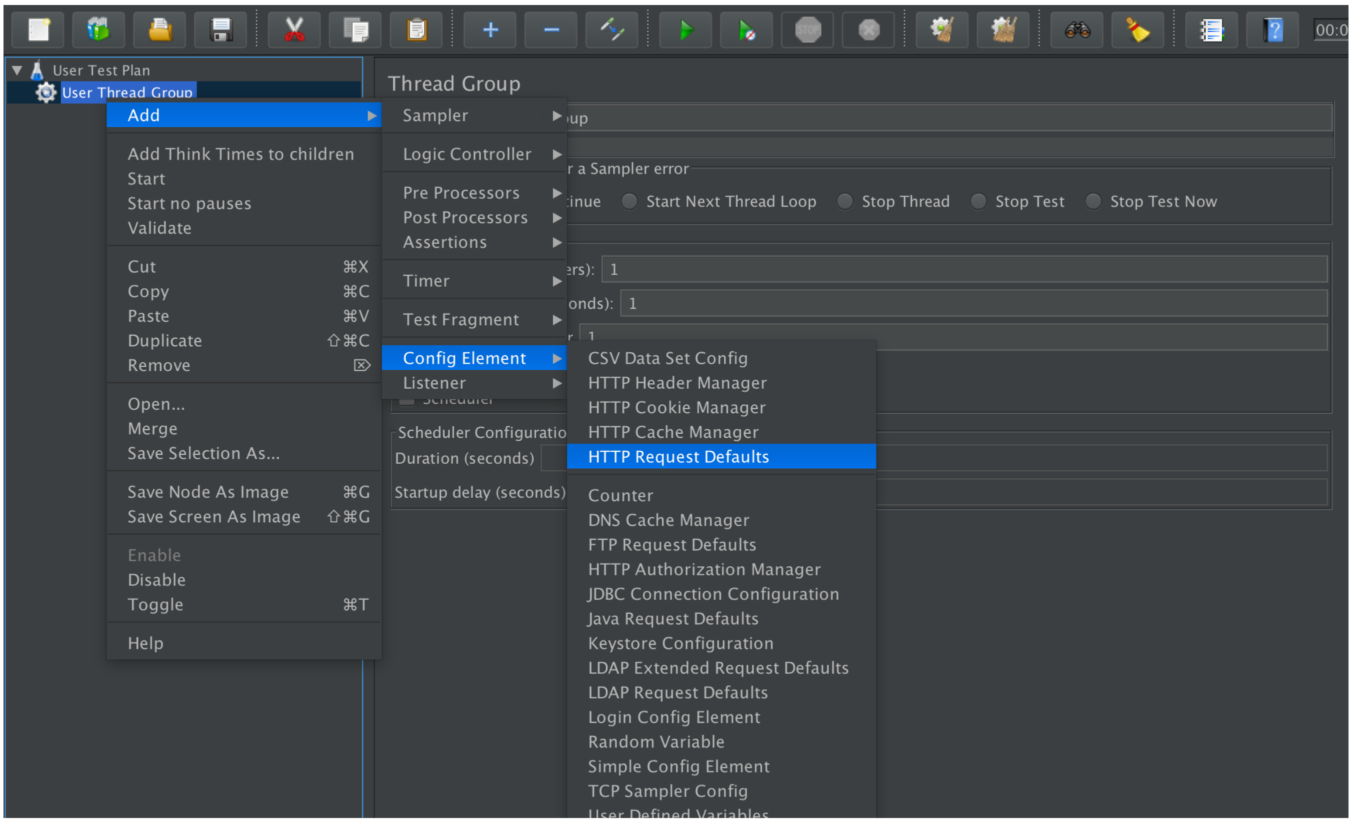
Let's take a real world example where we are going to create a JMeter test plan with a thread group which can run CRUD operations on user resource at a given server. Each "create user" operation needs to have different email address randomly generated before sending the request. Fetch authentication token and user id provided a response to create user request. An authentication token is used as the header in all subsequent requests and user id in URL and body parameter as needed. Finally, have a view result tree listener where we can see all requests with their responses.

Step 1: Add a Thread Group to our test plan



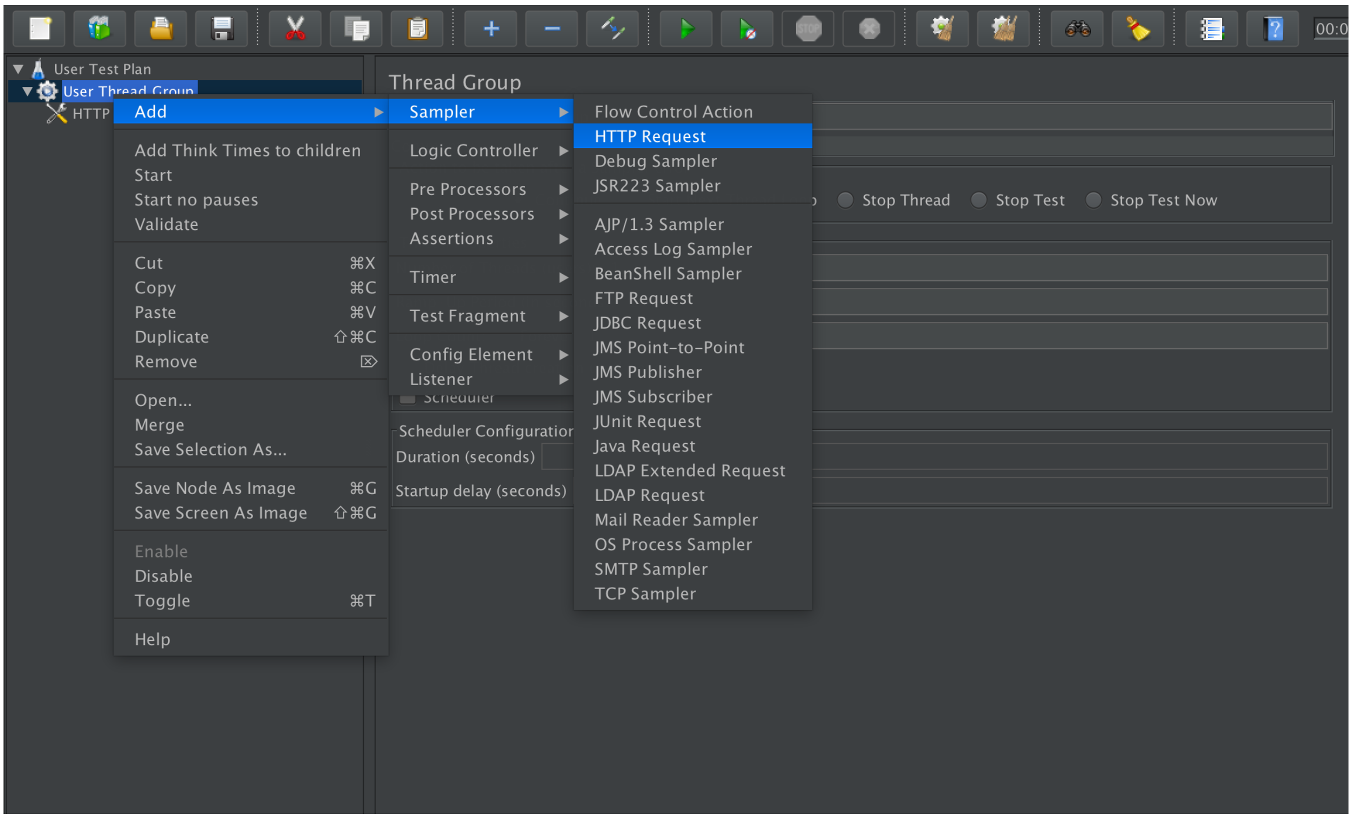
Creating a Thread Group

Step 2: Add an HTTP request defaults using the config elements to define default parameters like hostname and port of server needed for all requests.



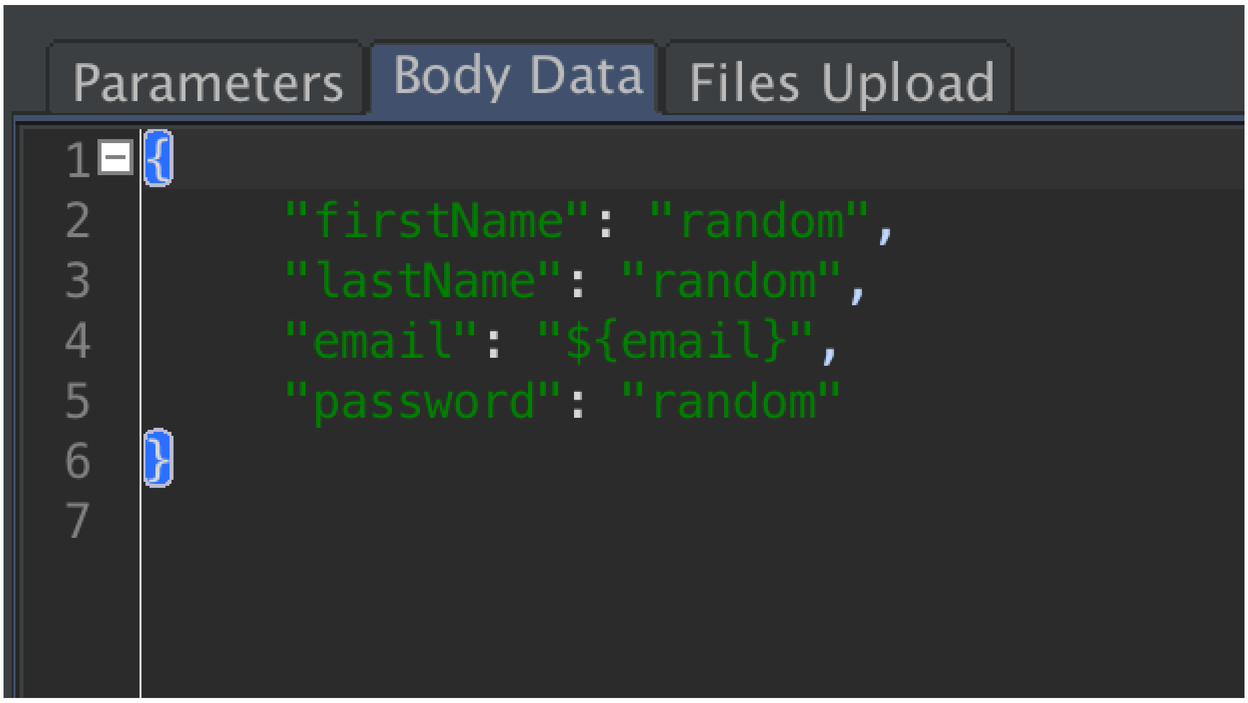
Creating HTTP Request Defaults

Step 3: Add an HTTP request sampler to send create user request.



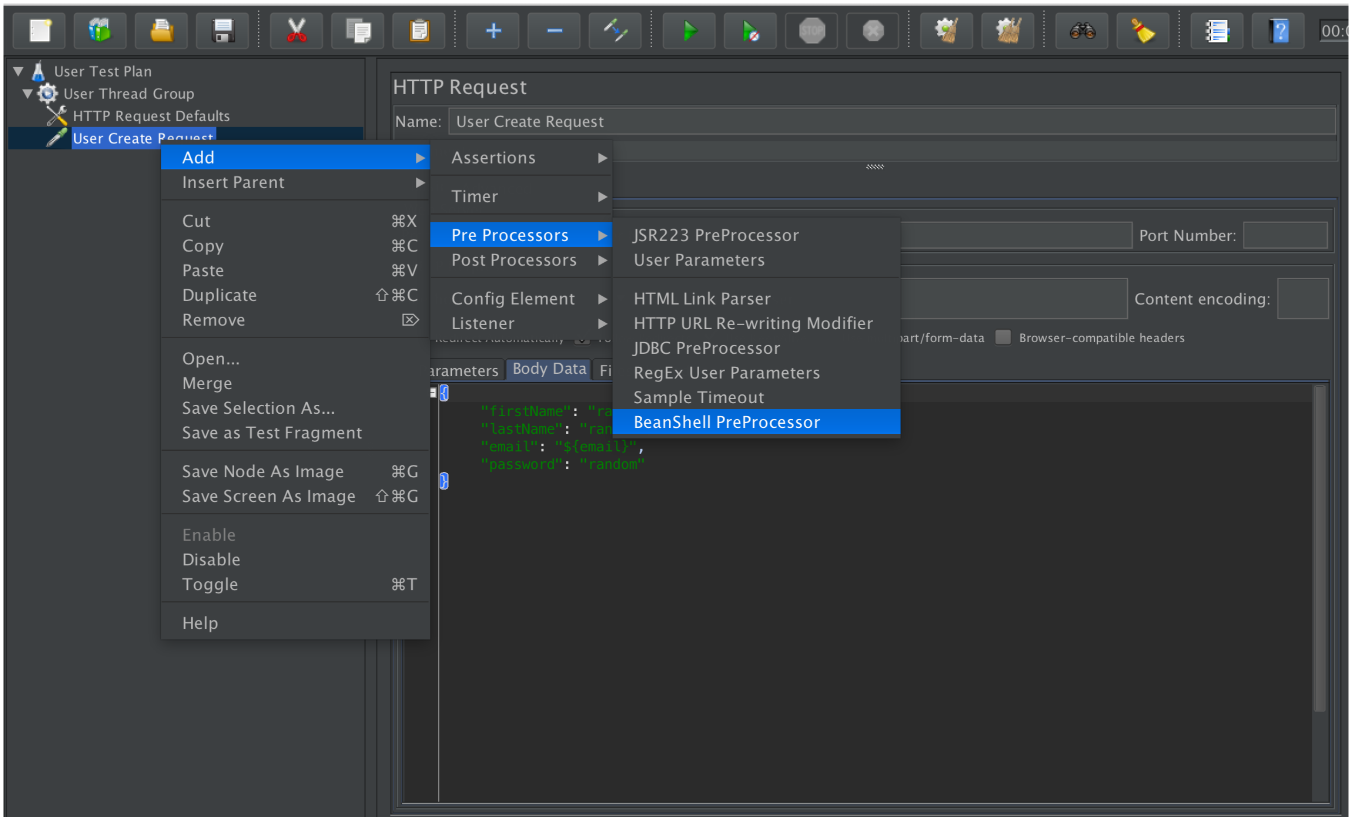
Creating HTTP Request sampler

Add body to create an HTTP request with a dynamic email parameter which is generated by pre-processor.



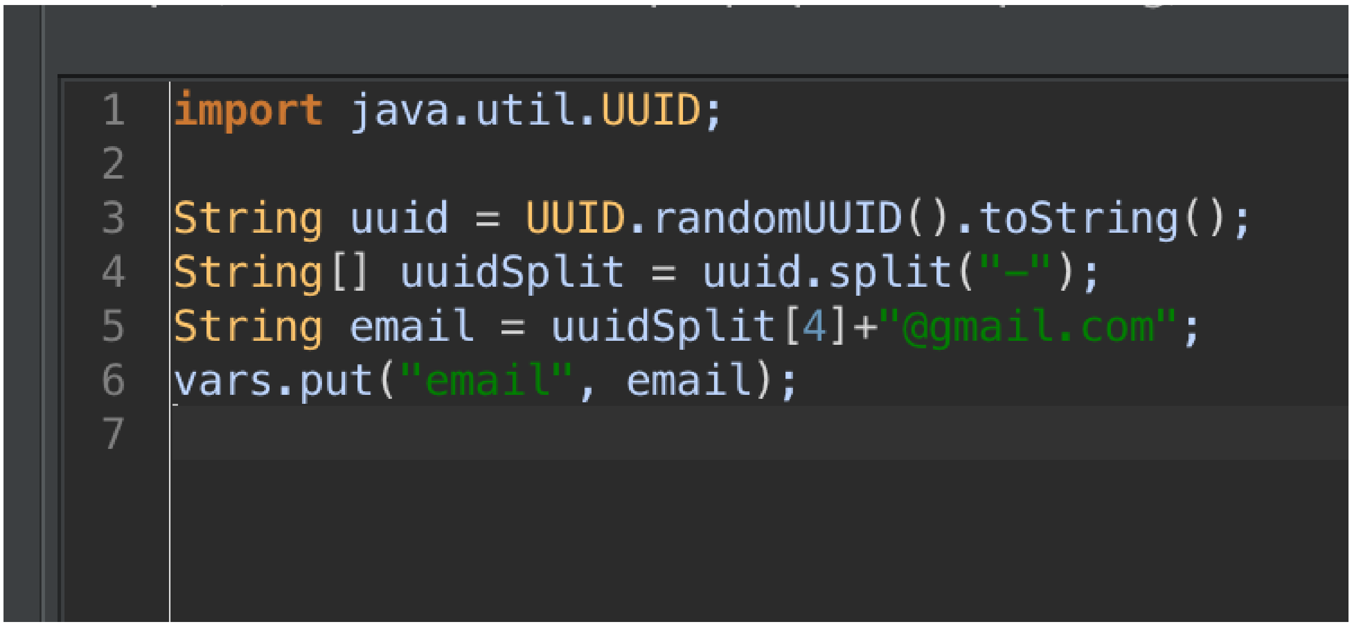
```
{
        "firstName": "random",
        "lastName": "random",
        "email": "${email}",
        "password": "random"
}
```

Step 4: Add a BeanShell pre-processor to generate a random email address for creating user request.



Creating BeanShell Processor for pre-processing of user HTTP Request

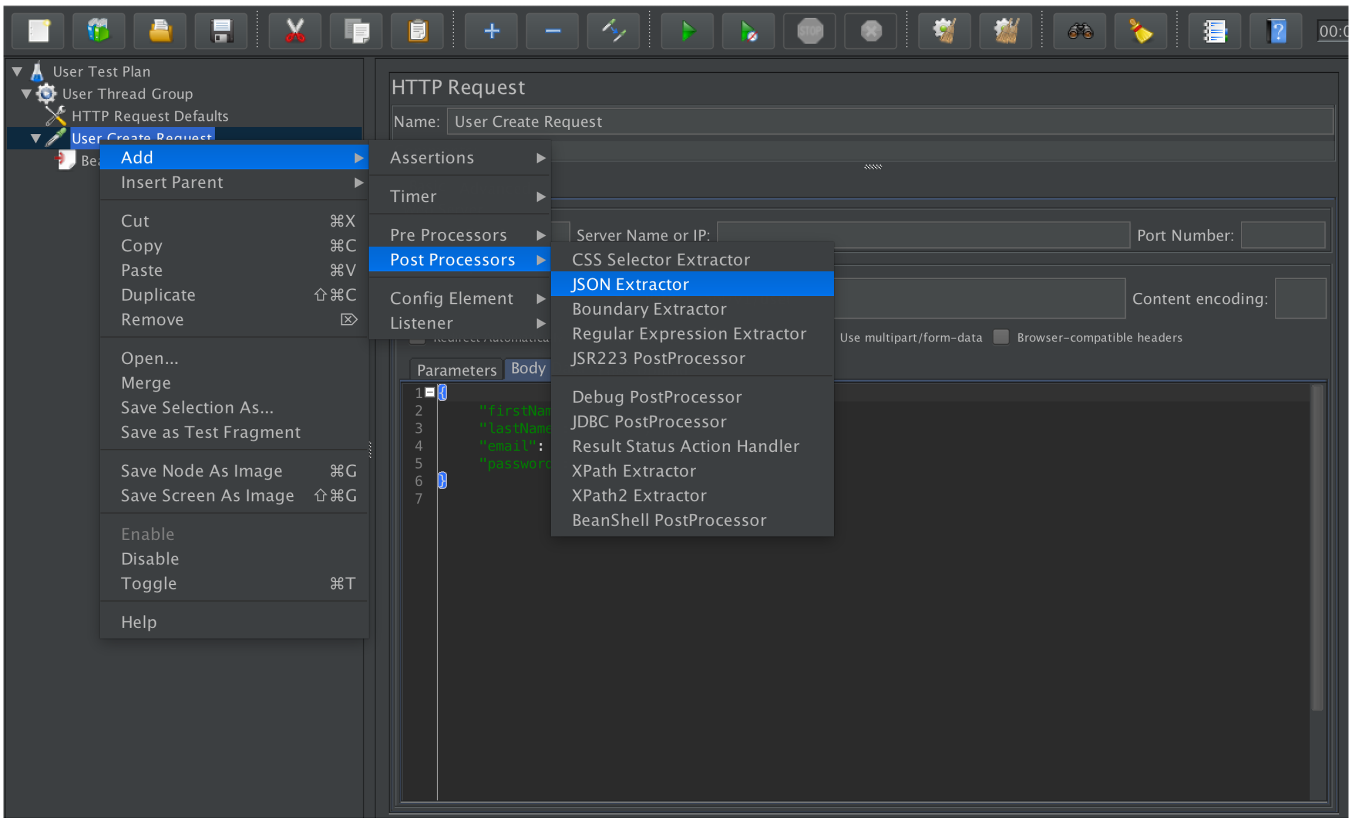
Using java code to generate a random number which is used in new email generation for each request.



```java
import java.util.UUID;

String uuid = UUID.randomUUID().toString();
String[] uuidSplit = uuid.split("-");
String email = uuidSplit[4]+"@gmail.com";
vars.put("email", email);
```

Step 5: Add a JSON extractor post-processor to get authentication token and userId from the response of creating user request.

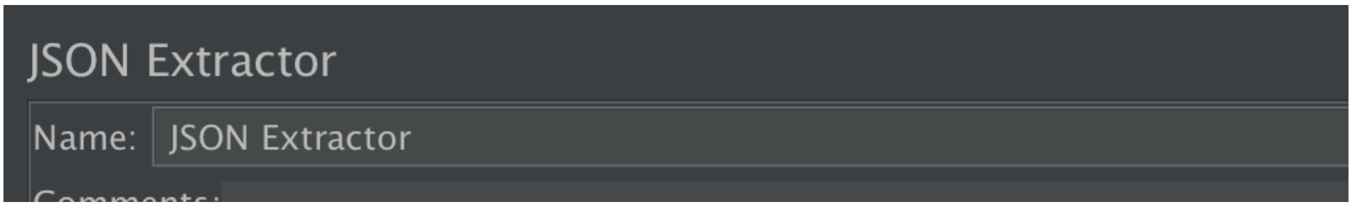


HTTP Request

Name: User Create Request

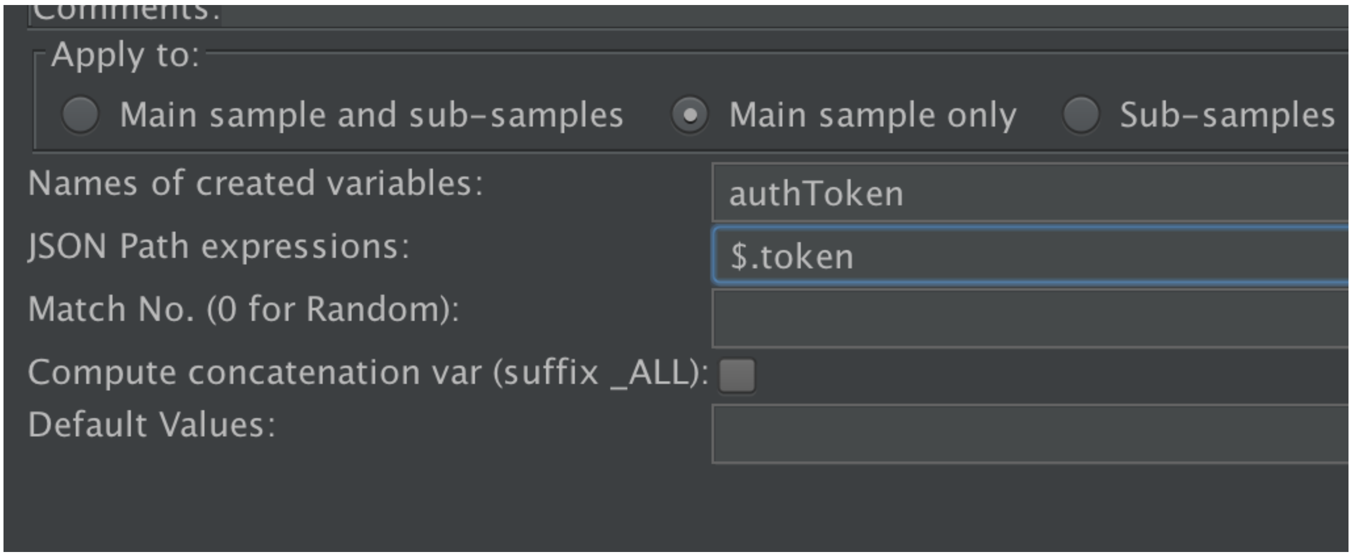
Creating JSON Extractor for post-processing HTTP request

Reading token from JSON response using JSON extractor.



JSON Extractor

Name: JSON Extractor


Apply to: Main sample and sub-samples [✓ Main sample only] Sub-samples

Names of created variables: authToken

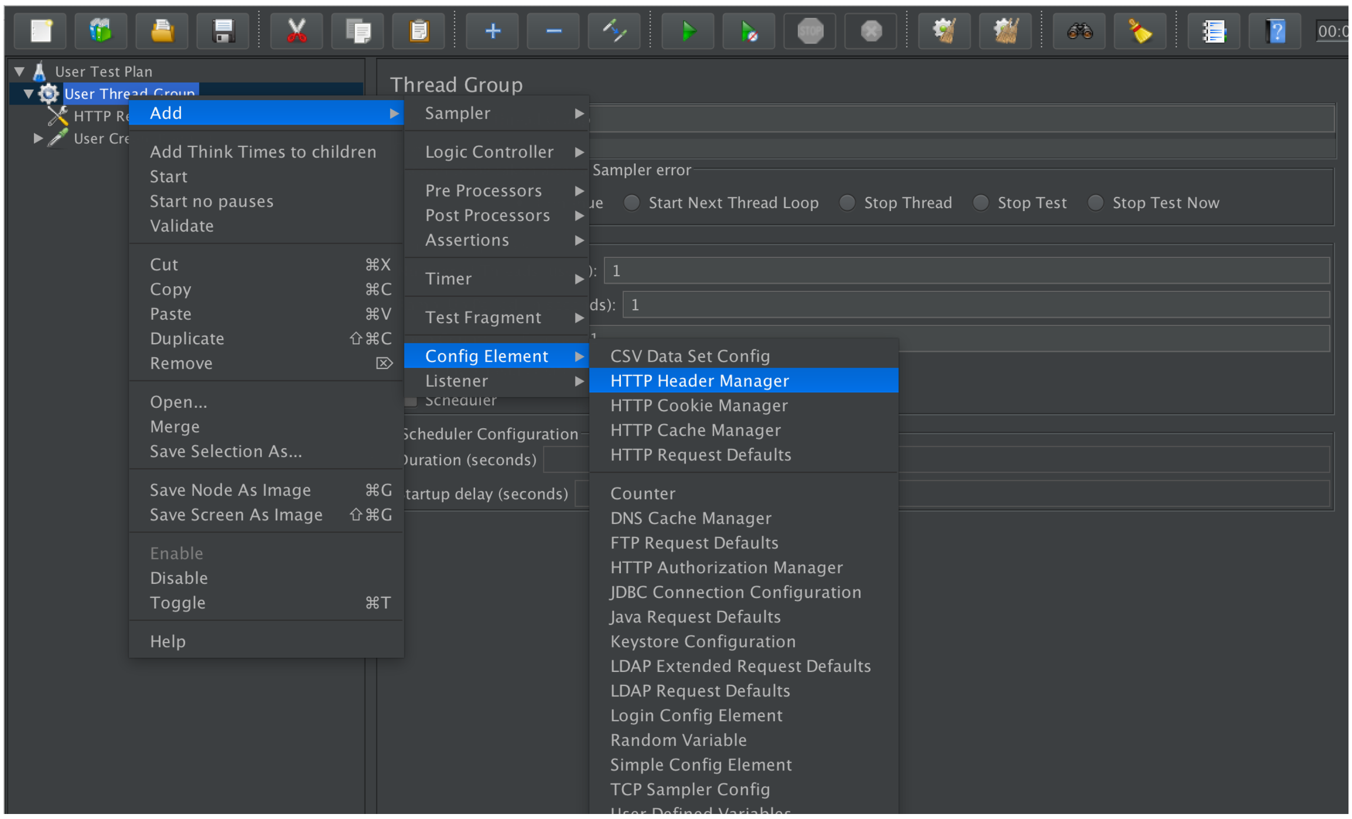
JSON Path expressions: $.token

Match No. (0 for Random):

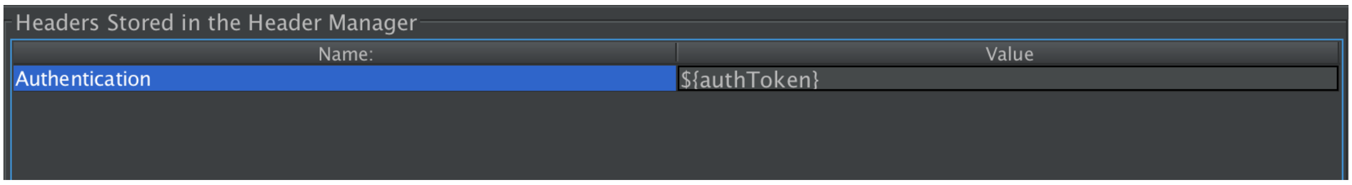
Compute concatenation var (suffix _ALL):

Default Values:

Step 6: Add an HTTP header manager to add authentication token as the header to all subsequent requests.



Creating HTTP Header Manager for sending auth header to all requests

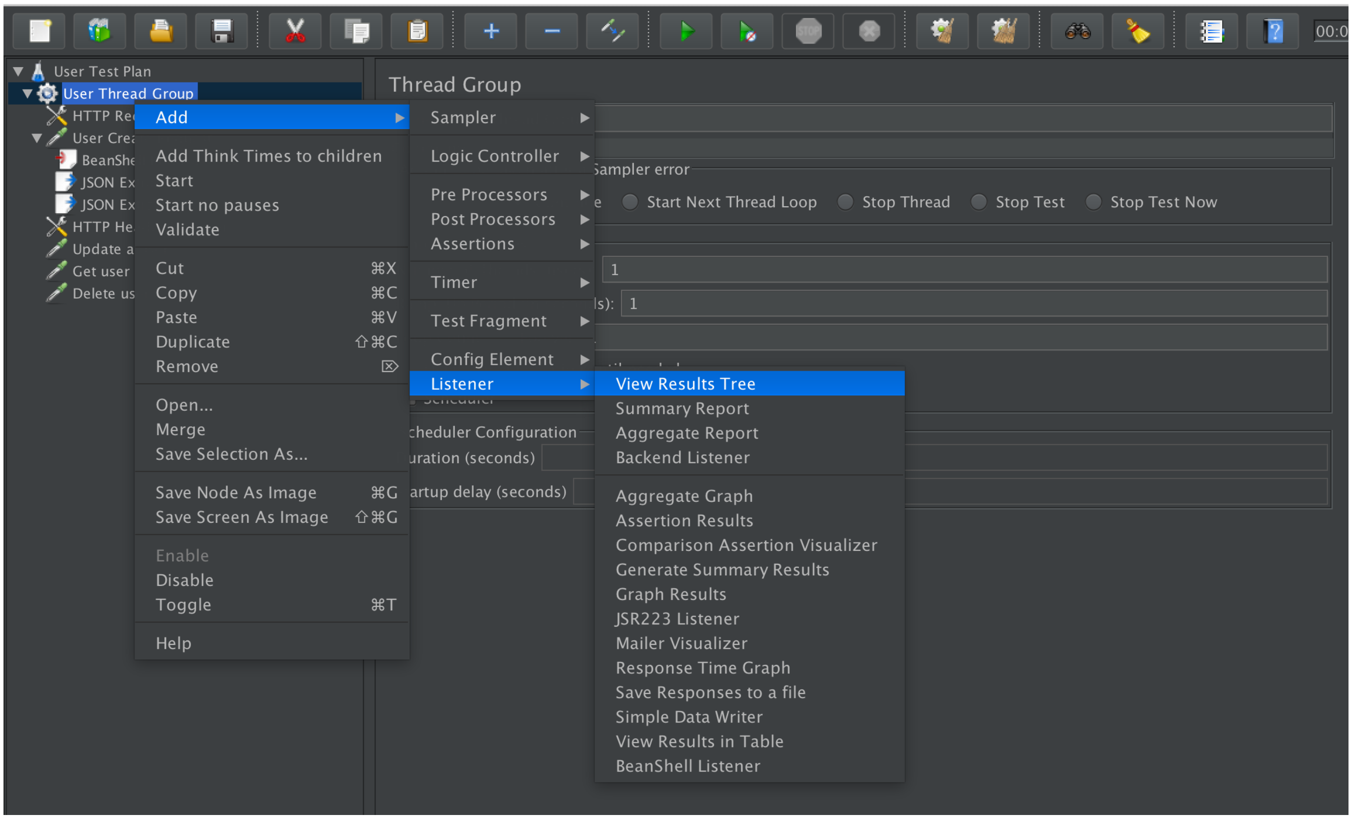

Headers Stored in the Header Manager

Name:	Value
Authentication	${authToken}

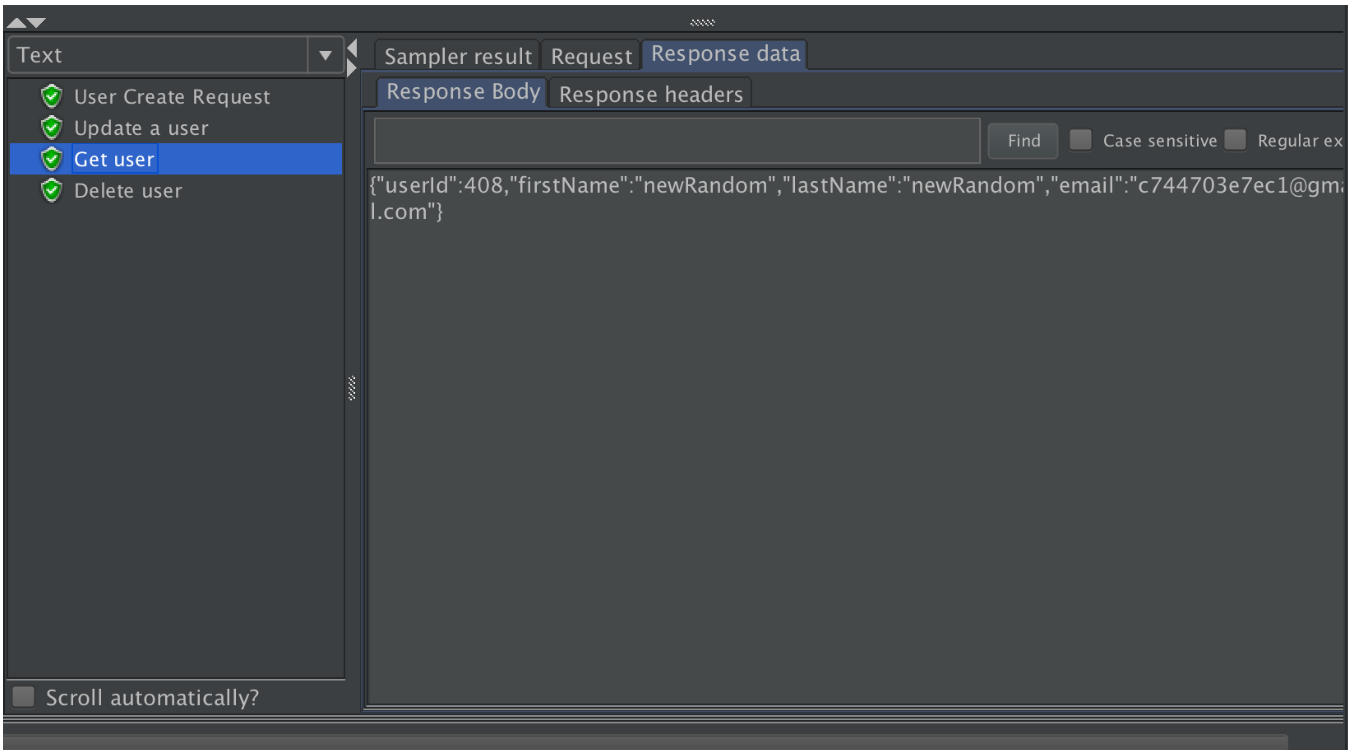
Step 7: Add update user, get user and delete user HTTP request sampler for subsequent requests in the same way as creating user HTTP request sampler with userId as dynamic

variable fetched from creating user post-processor.

Step 8: Add View result tree to get the response to all requests.

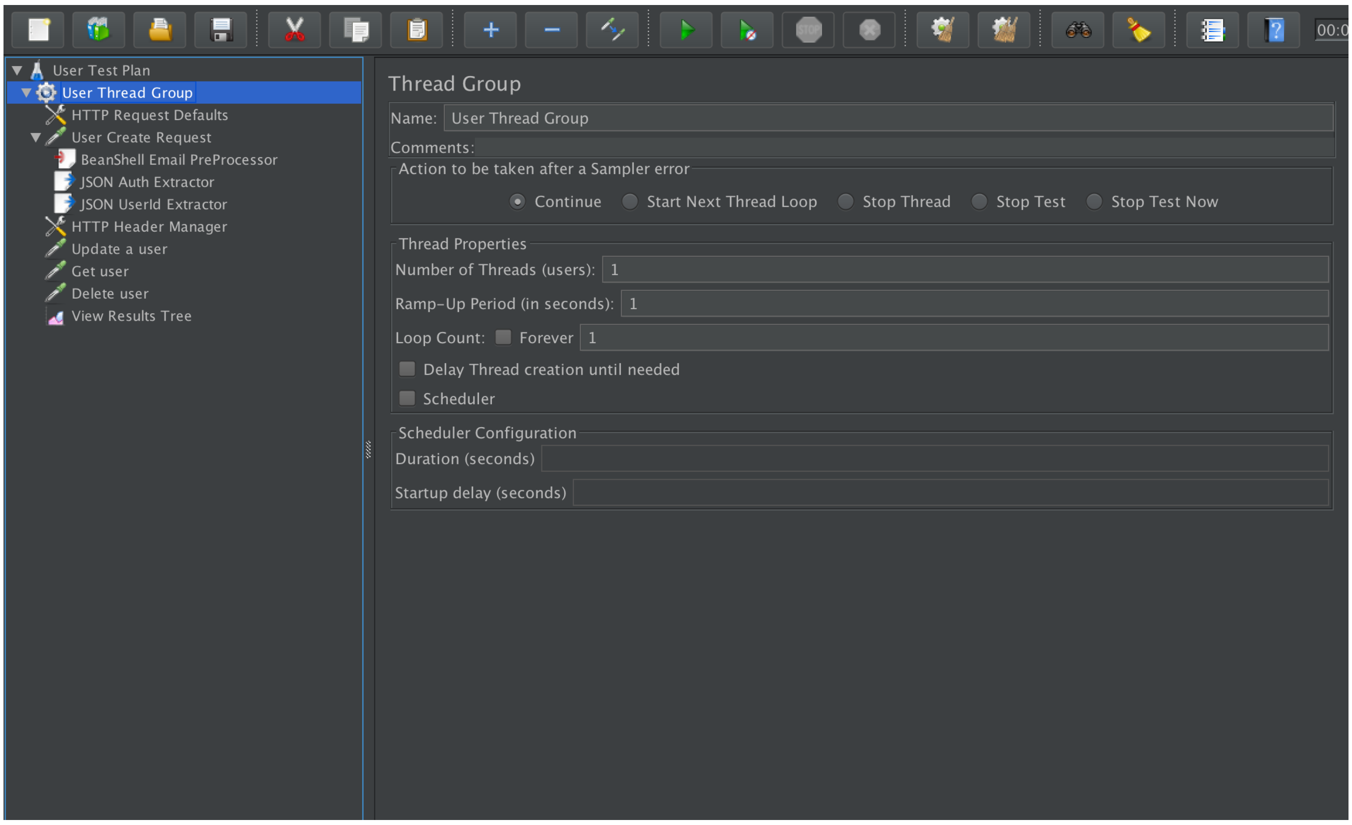


Creating View Result Tree Listener



```
{"userId":408,"firstName":"newRandom","lastName":"newRandom","email":"c744703e7ec1@gmai
l.com"}
```

Finally, we have our user CRUD load test ready which can be used to generate load by increasing the number of users with ramp-up time.



User Test Plan
- User Thread Group
  - HTTP Request Defaults
  - User Create Request
    - BeanShell Email PreProcessor
    - JSON Auth Extractor
    - JSON UserId Extractor
  - HTTP Header Manager
  - Update a user
  - Get user
  - Delete user
  - View Results Tree

Thread Group

Name: User Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 1

Ramp-Up Period (in seconds): 1

Loop Count: Forever 1

Delay Thread creation until needed

Scheduler

Scheduler Configuration

Duration (seconds)

Startup delay (seconds)

This way we can generate different load test plans to know the limits of our applications.

The complete code can be found in this git repository: https://github.com/MiteshSharma/JMeterUserLoadTest

PS: If you liked the article, please support it with claps 👏. Cheers