Stress Testing Guide

Version 1.0 (Jan 26)

© Pyramid Analytics BV 2026

Table of Contents

1. Overview 1

1.1. Key Ingredients 1

1.2. Test Plans 1

1.3. Intelligent Test Seeding 1

2. General Setup 3

2.1. Repository backup 3

2.2. Scaling and Hardware 3

2.3. Internal Settings 3

2.4. Logging 5

2.5. Users and access 6

2.6. Authentication 6

2.7. Other Considerations 7

3. Testing with JMeter 8

3.1. JMeter skeleton overview 9

3.2. Initial Configuration 10

3.3. Thread Group config 12

3.4. User Authentication 16

3.5. Recording Controller and Results 19

4. Recording a Test with JMeter 22

4.1. Overview 22

4.2. Planning the test 22

4.3. Test Recording Setup in JMeter 23

4.4. Recording the test 25

4.5. Recording cleanup 27

5. Running JMeter Tests 28

5.1. Configure test 28

5.2. Running the test 28

6. Analyzing Test Results 29

6.1. User total scenario time 29

6.2. Sub-Scenario time 29

1. Overview

This document outlines an approach to stress testing the Pyramid platform using tools like Apache JMeter. Given the inherent complexity of the product and dynamic behavior of the system - a simple approach to stress testing is mostly insufficient. Instead, a methodical, adaptive strategy must be applied - one that accounts for both the technical limitations of automated tools and the nuanced workflows inherent to Pyramid. Although JMeter was used in this document, the concepts discussed can be applied to other testing tools and frameworks.

1.1. Key Ingredients

Preparation is foundational to the success of any performance evaluation. It begins with ensuring that the test environment is suitable, including the configuration of external dependencies, services, and user accounts.

Stress testing the Pyramid platform may be influenced by external components. Key factors include the capacity and configuration of external databases, which must support high load, connection limits, and potential request throttling. Network infrastructure can introduce latency or bandwidth constraints, especially when interacting with external services like email, cloud storage, authentication systems, or other Pyramid nodes. Additional components such as reverse proxies, load balancers, or co-located applications can also impact performance. The hardware and network capacity of the testing application (like JMeter) must be considered - running tests from a single agent machine can create a bottleneck, so distributing the load across multiple agent machines is recommended for large-scale tests.

1.2. Test Plans

One of the primary considerations is the importance of defining realistic load scenarios - test cases that mirror actual production behavior in terms of user interaction, request pacing, and data dependencies. Unrealistic or overly synthetic scenarios may stress the system in ways that do not reflect real-world usage, leading to false conclusions about system limits or performance regressions.

It is strongly recommended to avoid state-changing or destructive operations in your test scenarios. Requests that create, update, or delete resources should not be used in the scenario flows to prevent unwanted side effects, such as database pollution or invalid application states. Where such state changes need testing, they should be designed to run using UI End-to-End tests and not via recorded request/response stress tests.

1.3. Intelligent Test Seeding

The Pyramid platform includes some advanced security and functional aspects that will create major problems when doing a simplistic **blind playback** of request/response tests. These will cause unnatural server responses, an unstable instance of Pyramid and ultimately bad testing results. It is therefore imperative that testing is seeded with proper users and sequences. To ensure consistency and access control during stress testing, the system must be preconfigured with enough test users, and they must all have appropriate access to shared content, including data sources, databases, and models - ideally through public folders with assigned roles.

In summary, this document provides a skeletal guide to planning, preparing, and executing stress tests. By combining sound preparation practices, realistic scenario design, and strong attention to environmental integrity and security, it enables teams to generate meaningful insights into system behavior when loaded and prepared for real-world scale.

2. General Setup

To run stress testing on a Pyramid platform the following items should first be addressed.

2.1. Repository backup

Executing non-supported operations (discussed in section 4.2.3) concurrently through a third-party application might cause an unknown database state that may/may not be recoverable. Therefore, before running any stress tests a database backup is suggested to avoid bigger data loss. Where possible, stress testing should be done on a separate system.

2.2. Scaling and Hardware

A central requirement is to ensure that the hardware and environment hosting the Pyramid platform can handle the expected load. This extends to the application engines and the in-memory database (if used). Refer to the Pyramid Scaling Guide for more information on best practices for setting up a scalable environment.

2.3. Internal Settings

There are a variety of internal settings that should be reviewed as part of the stress testing setup.

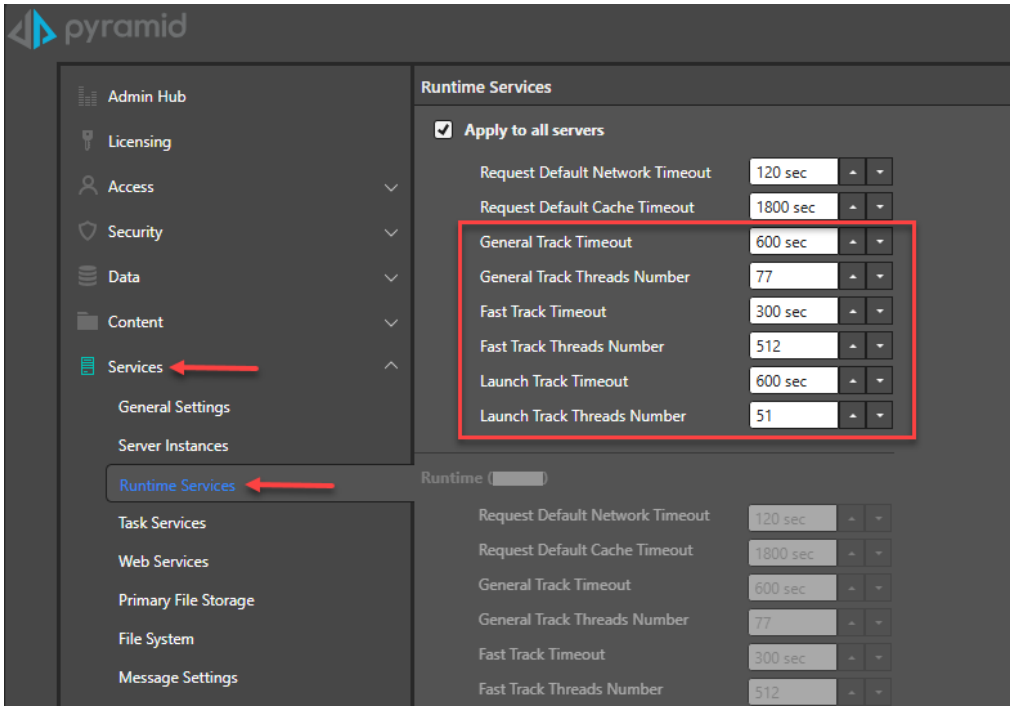
2.3.1. Runtime services settings

If you're testing live query performance (the most typical test scenario), configuring the Runtime Services settings is recommended.

Different requests in Pyramid are handled by different threading tracks in the system as described here. The thread count for each track affects the number of requests that can run at the same time for each run time engine. Setting the threads correctly can greatly improve outcomes.

- If these are set too high, thread thrashing will cause each request to fight for resources with other concurrent requests slowing CPU performance and even causing OOM (Out of Memory) issues.
- If these are set to low, the setup may throttle requests processing leading to underutilization of your resources and slower performance.

Note that if the same machine has other servers like the Task Engine, the available resources to handle requests will compete and make performance harder to achieve.



2.3.2. Timeouts

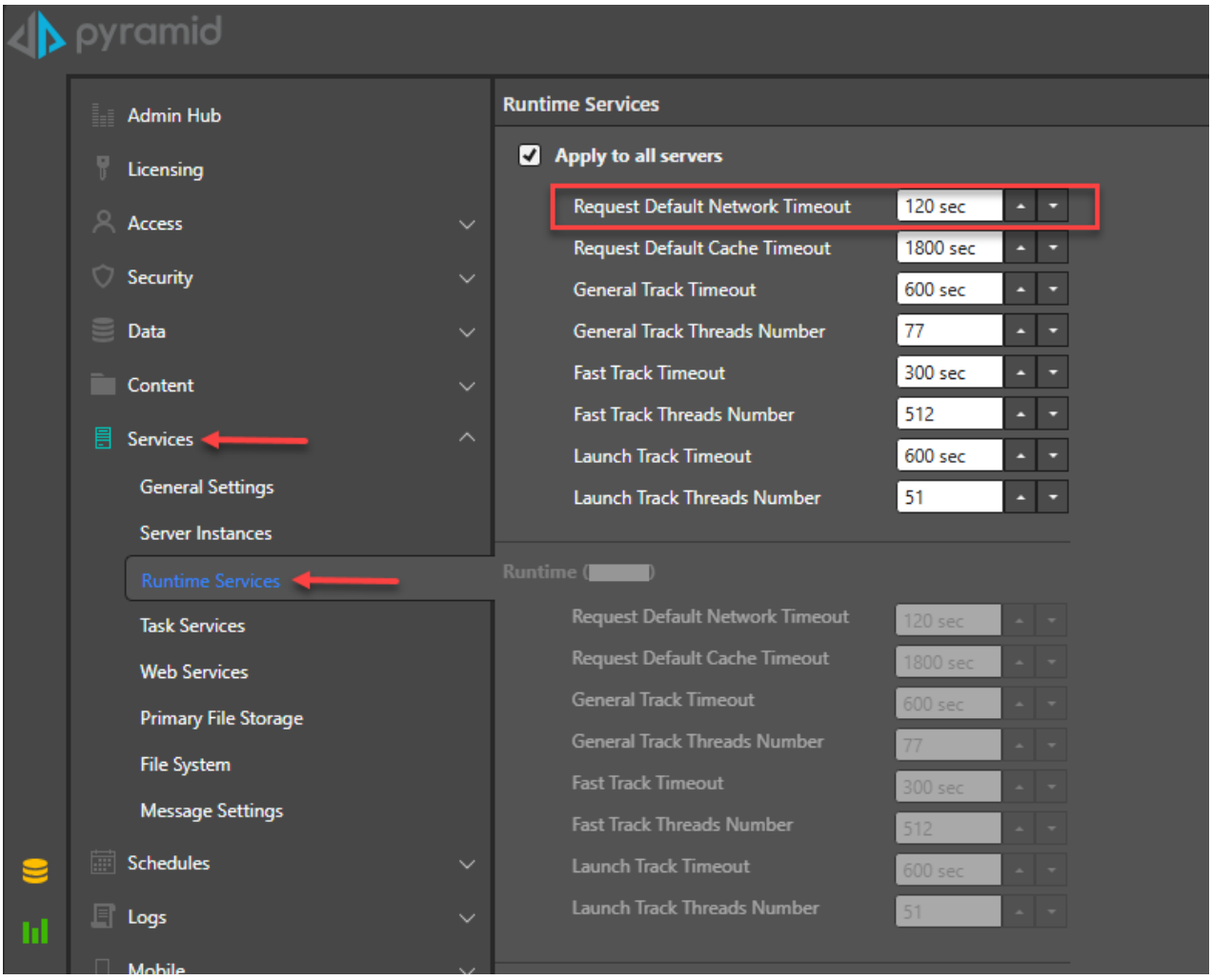
Network settings need to accommodate the expected stress from having hundreds or thousands of requests and responses flowing the system.

Long requests may be terminated due to network timeouts in their path (reverse proxy, load balancer, firewalls).

After a long request has been timed out, the web client sends additional requests to retrieve the response of the previous request.

If the timeout settings are not long enough, additional logic may be needed to make sure that a request is finished before the next request can be sent.

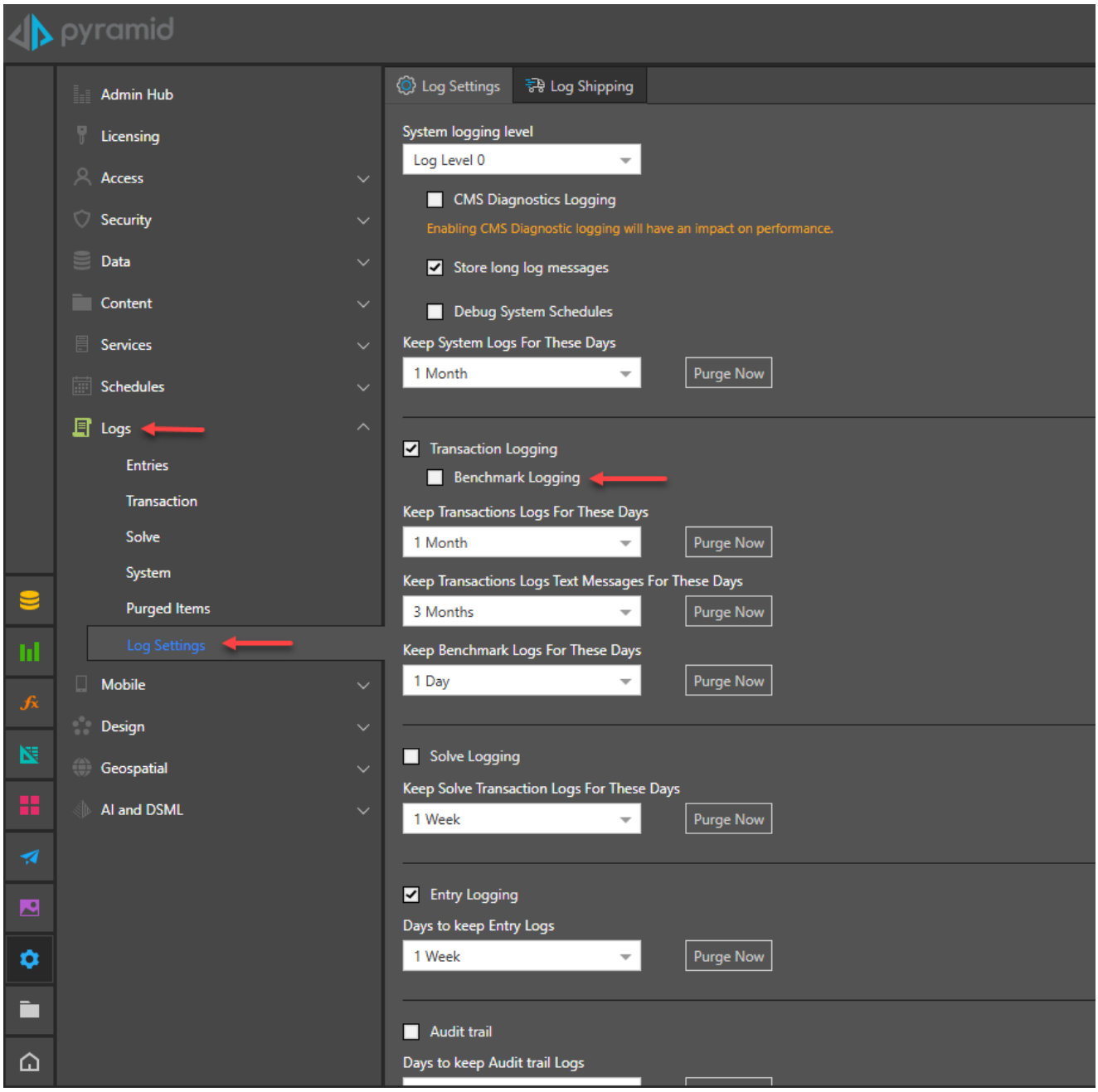
The settings of all middlemen network tiers need to be set independently from the Pyramid setting.



2.4. Logging

Logging in general affects the performance of the system due to the amount or R/W operations with the internal repository. Disabling unnecessary logging (high log levels or specific logs) is therefore encouraged.

- Benchmark logging should be turned off.
- System Logging should be at level 0
- CMS Diagnostics should be disabled



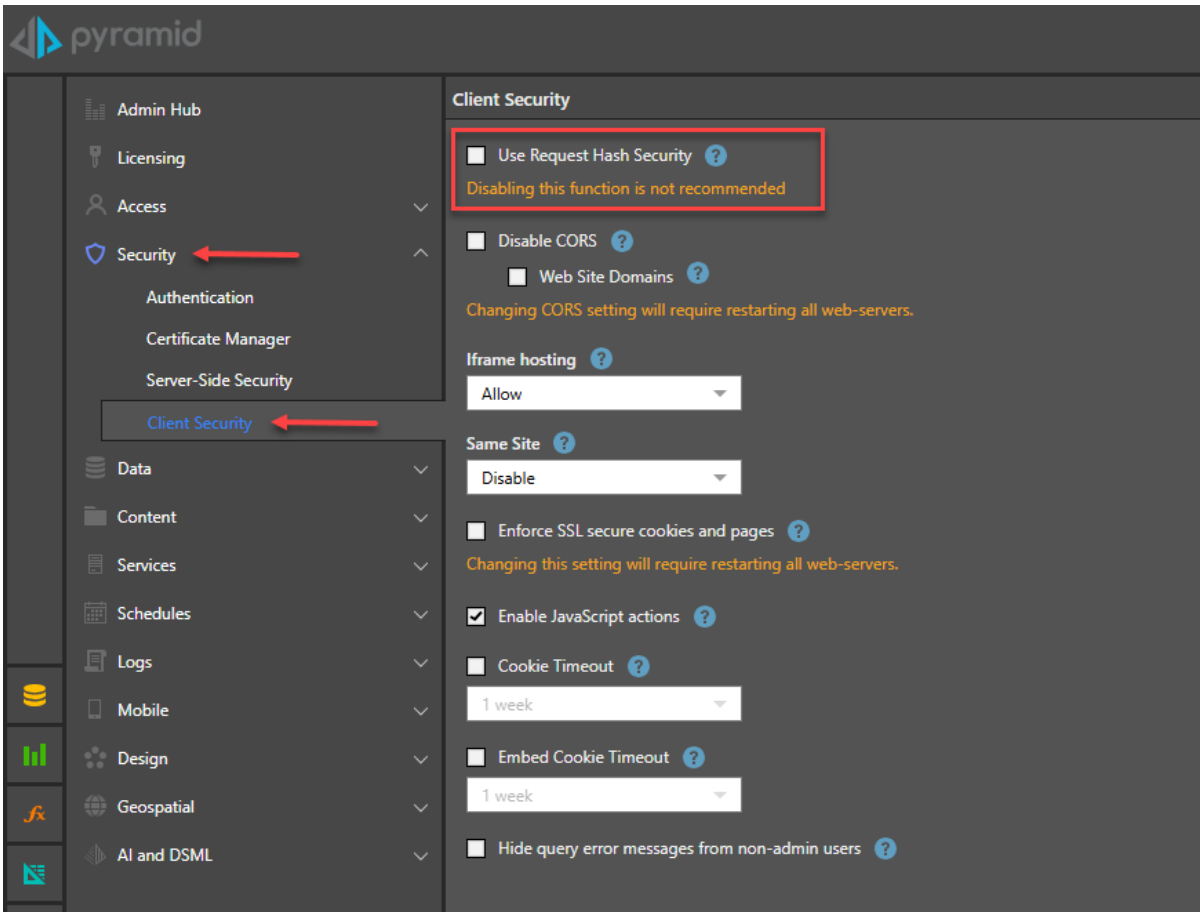
2.5. Users and access

Make sure the application contains different test users to run the scenarios needed. Reusing the same user can create unintended effects with caching, routing, and security operations.

All users recording and running the tests should be either of type “Enterprise Admin”, to avoid all the complexities of the built-in security hashing or non-admin tests users need to have the correct security, as described below.

2.5.1. Using Non-Admin users

The “Request Hash Security” setting should be turned off, to allow reuse of the same request for different users. Failure to disable this function will create numerous internal errors, over processing/logging of issues and false response results.



All content should be accessible to all participating users, preferably to use the public folder with roles for all participating users.

All data sources, databases, models should be accessible to all participating users.

Note: In this document we assume all users’ usernames are in the following format: “TestUser<ID>”. where <ID> is a continuous number from 1 to the number of needed users [e.g. TestUser1, TestUser2, …, TestUser100]. The “TestUser” prefix can be altered and defined later (3.2.2).

2.6. Authentication

The example in this document assumes that the system uses the internal “Database” authentication. Other authentication types can be tested and are described section 3.4.1, but they typically require more complex setups. While authentication testing is important, there is little difference between database authentication and other authentication mechanisms when stress testing Pyramid’s engines.

Use of “Windows Authentication” required additional configuration of the JMeter to impersonate the users and is outside of the scope of this document. It is NOT recommended.

2.7. Other Considerations

Testing of Pyramid platform can be affected by multiple factors outside of the scope of the application that need to be considered.

- **Analytic databases** – Day to day scenarios is usually distributed on different data sources, using specific scenario can burden this server, these are few points to note:
    - Resources - The data source used in the testing scenario must have enough resources to handle the amount of load it will receive.
    - Connection limit – Some serves limit the number of connections that can exist in any given time, this number needs to be considered.
    - Request throttling – Some external web services (e.g. google) can limit the number of requests that can be sent to it in each time period based on the account tier.
- **Peripheral services** – Any request sent outside of Pyramid is passing through the network infrastructure and other services. This can affect the performance due to delays or bandwidth limits. Such services need to be resourced properly or potentially eliminated from the testing agenda to get a clear reading of the core platform's operations. Examples include:
    - Data sources.
    - Email or SMS service.
    - External storage (e.g. Amazon S3).
    - Pyramid Pulse server.
    - Pyramid inter-server communication between nodes.
    - External Authentication services (e.g. LDAP, SAML, OpenID)
    - Additional applications that are installed on the nodes beside the Pyramid platform.
- **Other Pipelines** – Additional services in the pipeline of requests and responses may throttle testing on Pyramid and need to be configured correctly:
    - Revers-Proxy
    - Load balancer
    - Firewalls
- **Testing agents** – note that if the testing runs from single machine, this machine is limited in the network I/O, CPU and memory. Handling high volumes of queries and transactions can be overwhelming and running them from a single machine can create throttle the tests themselves. For high user/test throughput consider distributing the load between multiple testing agent machines, running concurrently.

3. Testing with JMeter

Key to stress testing is the use of a stress testing application. There are numerous choices in the market and most use a common approach with variations in function and setup. This guide was created using Apache JMeter version 5.5 – which is widely used for request/response stress testing of web applications.

This section will describe how to create a skeletal test plan with JMeter. This test plan will not contain any testing scenarios and can be reused for any specific testing later.

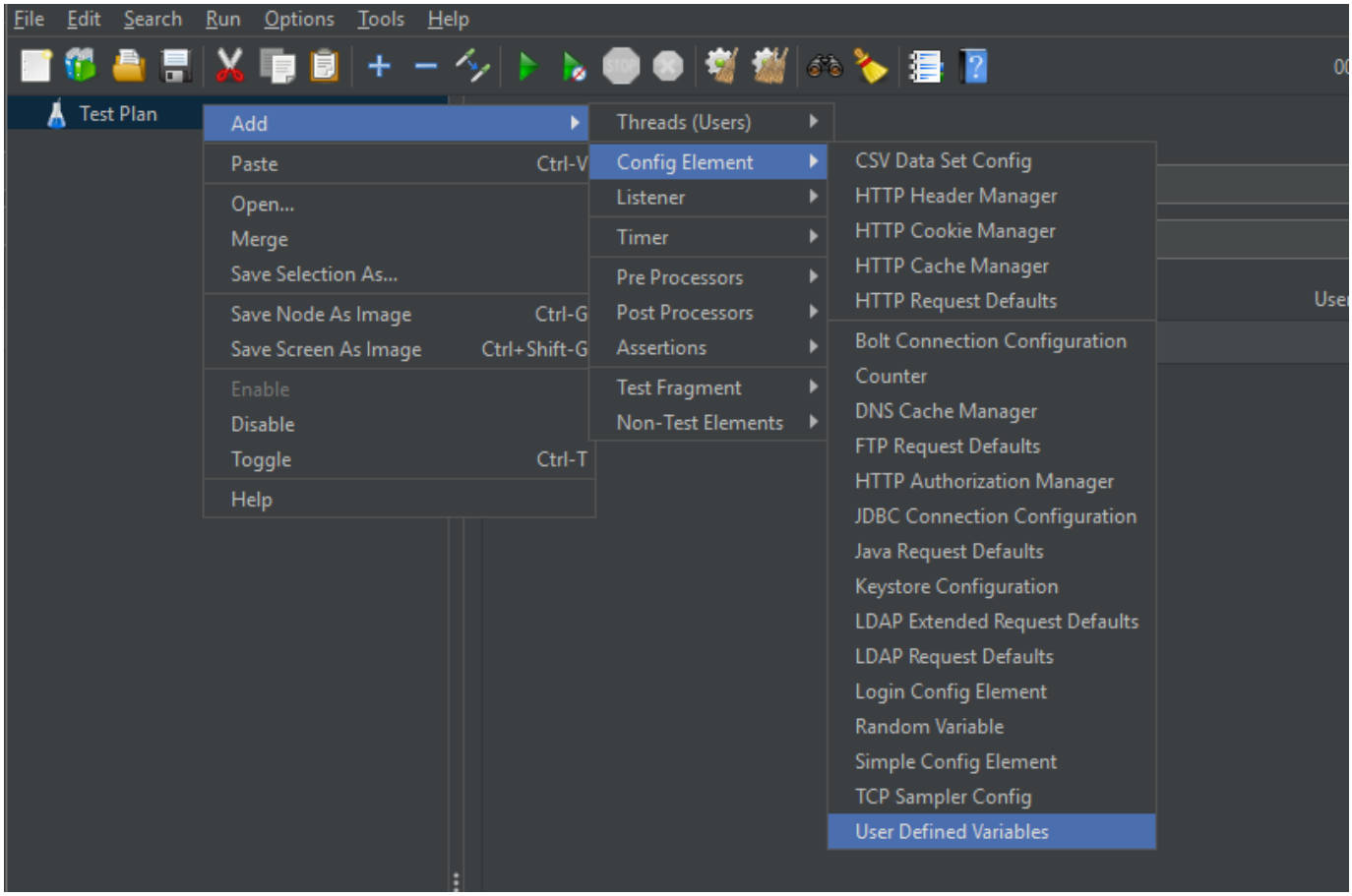
The use of this test plan will be described below.

Note: The following syntax will be used for adding a node into the test plan In JMeter

[Right-click on XXXX -> Path -> To -> Element]

e.g. adding an “User Defined Variables” node:

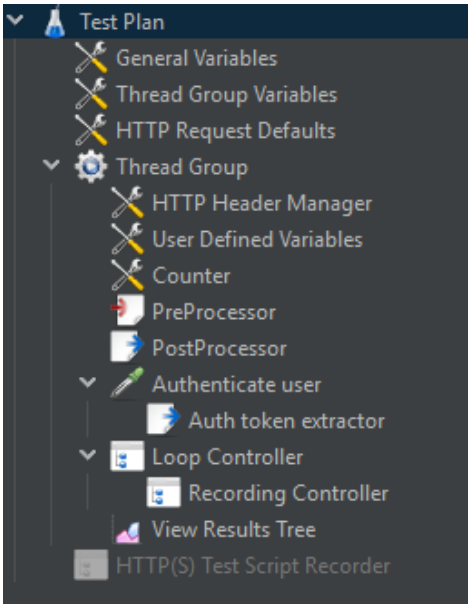
[Right-click on Test Plan-> Add -> Config Element -> User Defined Variables]



3.1. JMeter skeleton overview

This is the overview of the skeleton test plan and setup required in JMeter.

Section	Name	Type
	Test Plan	Test Plan
3.2.1	General Variables	User Defined Variables
3.2.2	Thread Group Variables	User Defined Variables
3.2.3	HTTP Request Defaults	HTTP Request Defaults
3.3.1	Thread Group	Thread Group
3.3.2	HTTP Header Manager	HTTP Header Manager
3.3.3	User Defined Variables	User Defined Variables
3.3.4	Counter	Counter
3.3.5	PreProcessor	JSR223 PreProcessor
3.3.6	PostProcessor	JSR223 PostProcessor
3.4.1	Authenticate user	HTTP Request
3.4.2	Auth token extractor	Regular Expression Extractor
3.5.1	Loop Controller	Loop Controller
3.5.2	Recording Controller	Recording Controller
3.5.3	View Results Tree	View Results Tree
3.5.4	HTTP(S) Test Script Recorder	HTTP(S) Test Script Recorder



3.2. Initial Configuration

Load up JMeter and then start with an empty test plan.

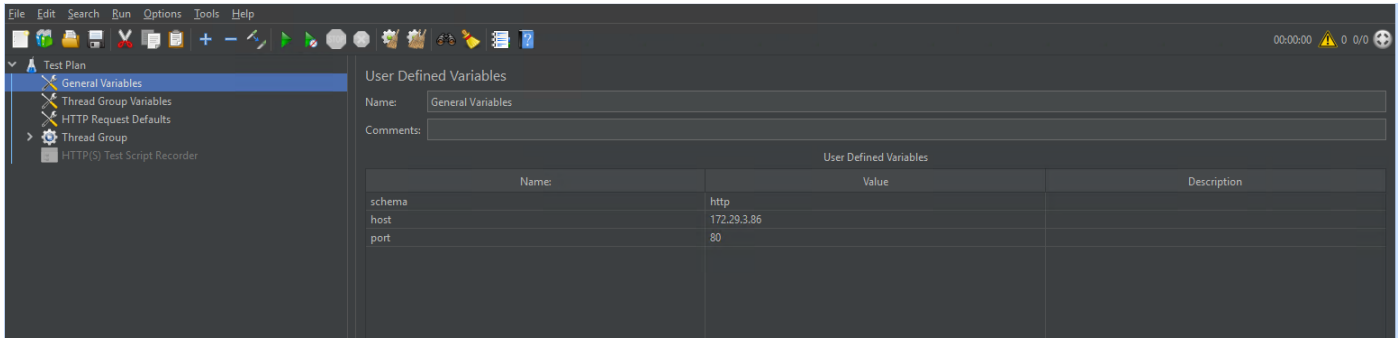
3.2.1. General Variables

Add a “User Defined Variables” node [Right-click on Test Plan-> Add -> Config Element -> User Defined Variables].

Fill the following:

- Name: General Variables
- Variables:

Name	Value	Descriptions
schema	http	Network scheme [http/https]
host	*Pyramid.host.com*	Hostname for the tested machine
port	80	Network port



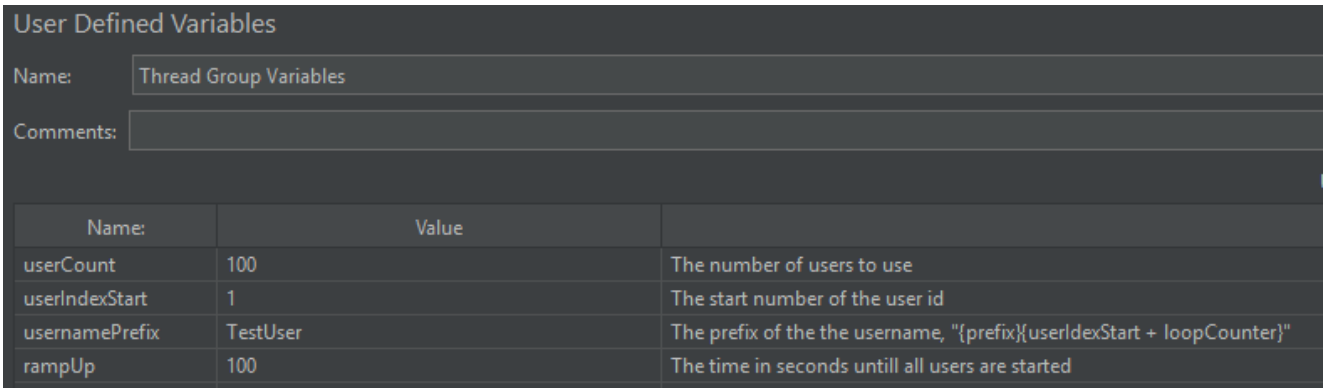
3.2.2. Thread Group Variables

Add a “User Defined Variables” node [Right-click on Test Plan-> Add -> Config Element -> User Defined Variables].

Fill the following:

- Name: Thread Group Variables
- Variables:

Name	Value	Descriptions
userCount	100	The number of users to use
userIndexStart	1	The start number of the user id
usernamePrefix	TestUser	The prefix of the username, "{prefix}{userIdexStart + loopCounter}"
rampUp	100	The time in seconds until all users are started

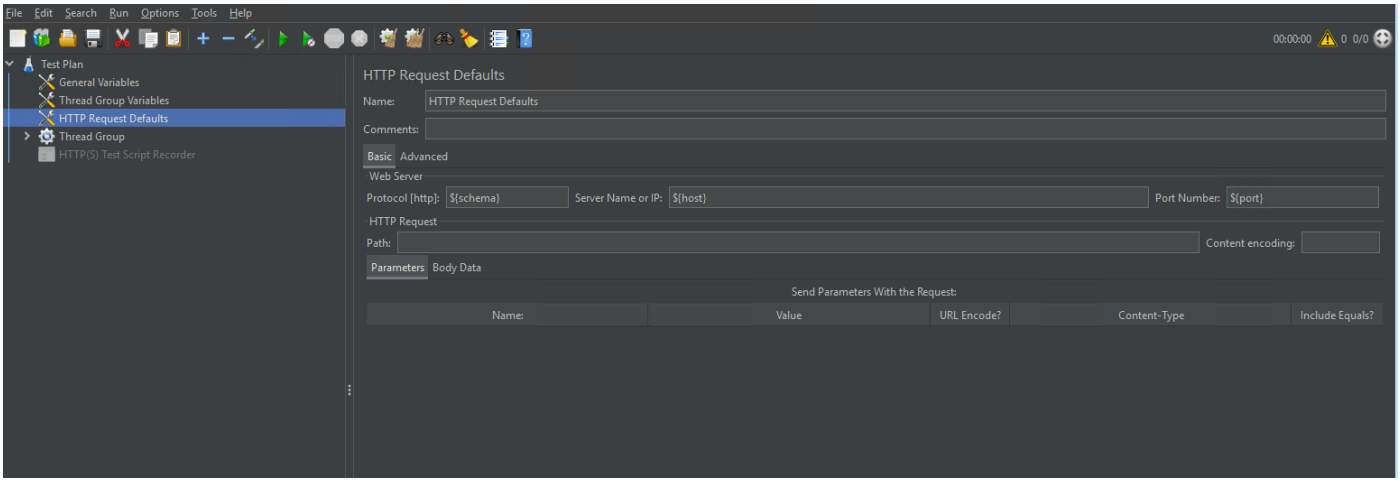


3.2.3. HTTP Request Defaults

Add a “HTTP Request Defaults” node [Right-click on Test Plan -> Add -> Config Element -> HTTP Request Defaults]

Fill the following:

- Protocol: ${schema}
- Server Name or IP: ${host}
- Port Number: ${port}



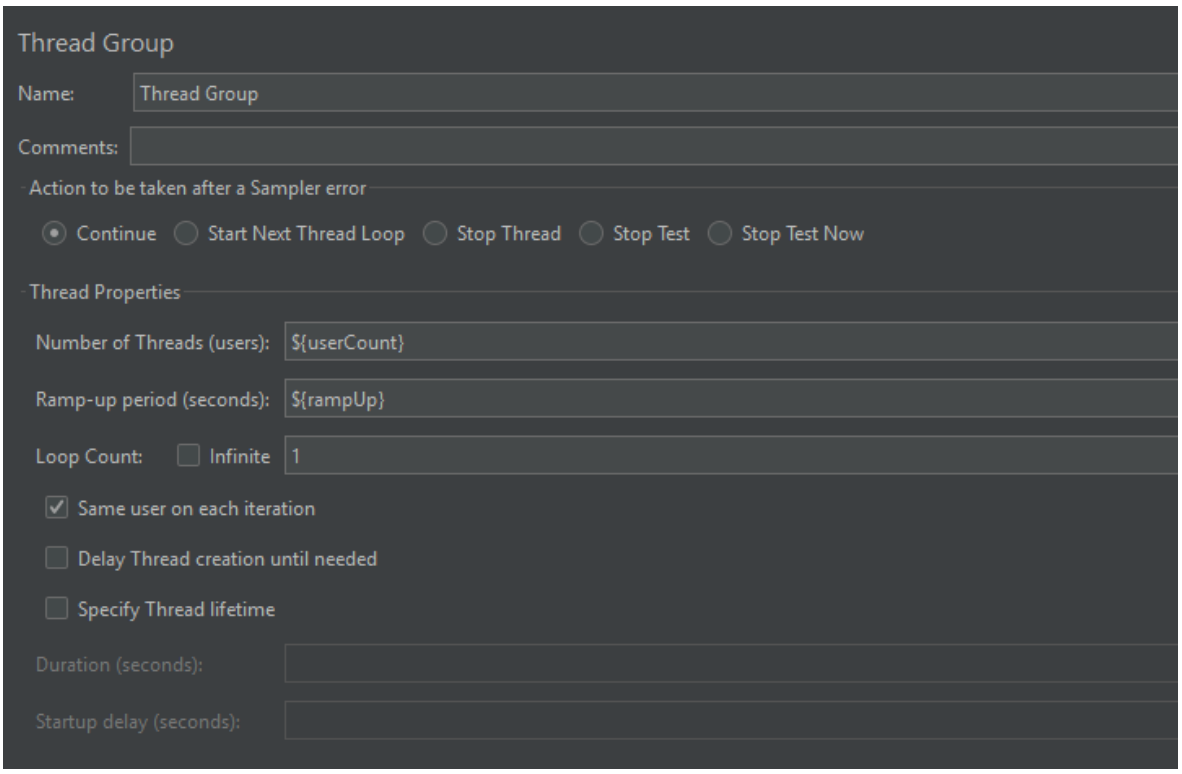
3.3. Thread Group config

3.3.1. Thread Group

Add a “Thread Group” node [Right-click on Test Plan -> Add -> Threads (Users) -> Thread Group]

Fill the following:

- Number of Threads (users): ${userCount}
- Ramp-up period (seconds): ${rampUp}

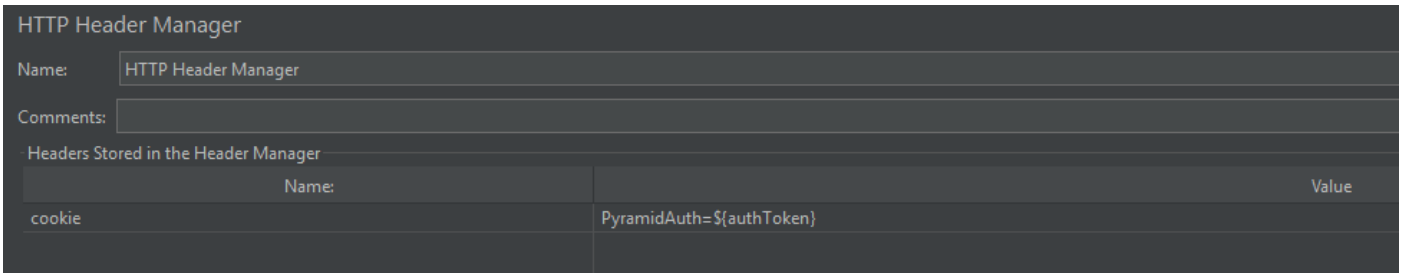


3.3.2. HTTP Header Manager

Add a "HTTP Header Manager" node [Right-click on **Thread Group**-> Add -> Config Element -> HTTP Header Manager]

Fill the following:

Name	Value
cookie	PyramidAuth=${authToken}

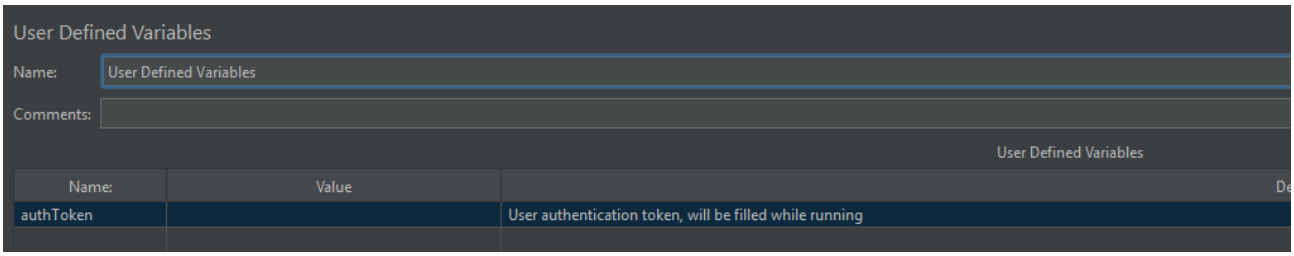


3.3.3. User Defined Variables

Add a "User Defined Variables" node [Right-click on **Thread Group**-> Add -> Config Element -> User Defined Variables]

Fill the following:

Name	Value	Descriptions
authToken		User authentication token, will be filled while running

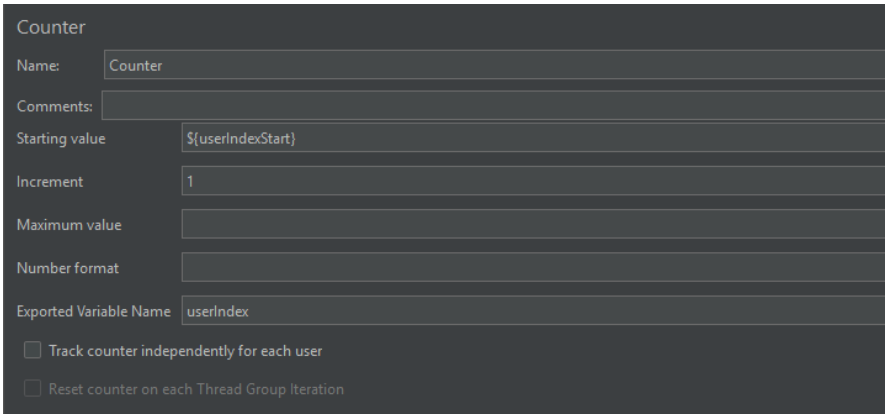


3.3.4. Counter

Add a "Counter" node [Right-click on **Thread Group**-> Add -> Config Element -> Counter]

Fill the following:

- Starting value: ${userIndexStart}
- Increment: 1
- Exported Variable Name: userIndex



3.3.5. PreProcessor

Add a "JSR223 PreProcessor" node [Right-click on **Thread Group**-> Add -> Pre Processors -> JSR223 PreProcessor]

Fill the following:

- Name: PreProcessor
- Script:

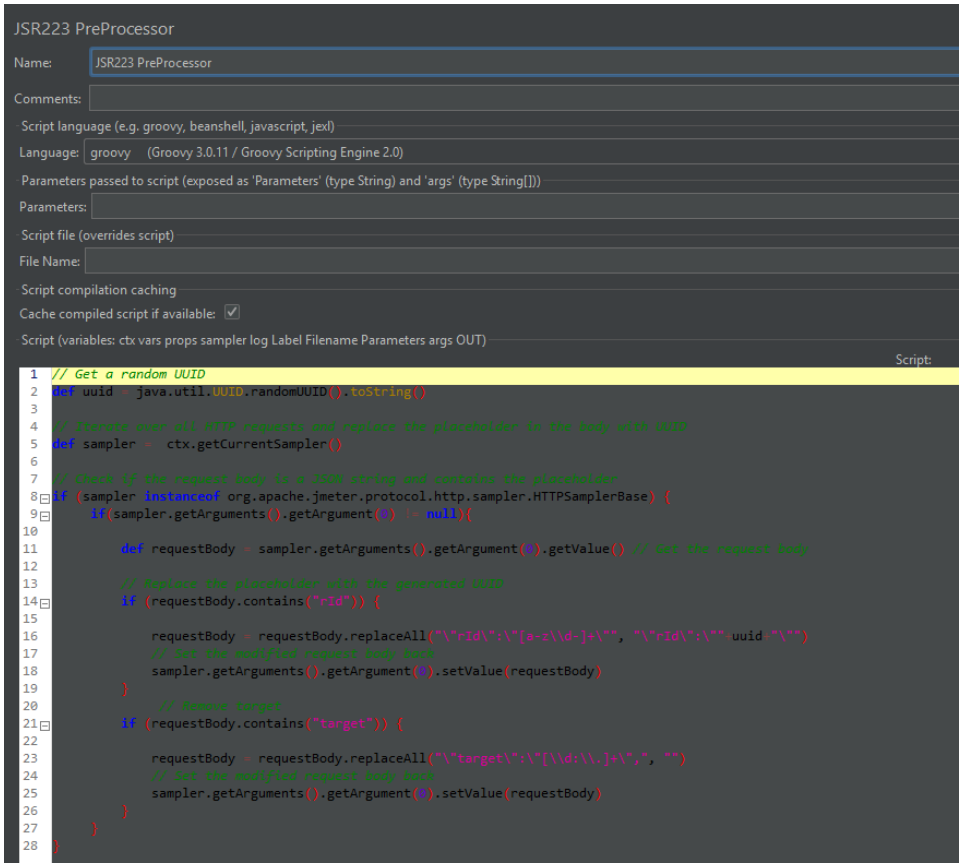
```
// Get a random UUID
def uuid = java.util.UUID.randomUUID().toString()

// Iterate over all HTTP requests and replace the placeholder in the body with UUID
def sampler =  ctx.getCurrentSampler()

// Check if the request body is a JSON string and contains the placeholder
if (sampler instanceof org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase) {
        if (sampler.getArguments().getArgument(0) != null){

            def requestBody = sampler.getArguments().getArgument(0).getValue() // Get the request body

            // Replace the placeholder with the generated UUID
            if (requestBody.contains("rId")) {
              requestBody = requestBody.replaceAll("\"rId\":\"[a-z\\d-]+\"", "\"rId\":\""+uuid+"\"")
              // Set the modified request body back
              sampler.getArguments().getArgument(0).setValue(requestBody)
            }
            // Remove target
            if (requestBody.contains("target")) {
              requestBody = requestBody.replaceAll("\"target\":\"[\\d:\\.]+\",", "")
              // Set the modified request body back
              sampler.getArguments().getArgument(0).setValue(requestBody)
            }
        }
}
```



3.3.6. PostProcessor

Add a "JSR223 PostProcessor" node [Right-click on **Thread Group**-> Add -> Post Processors -> JSR223 PostProcessor]

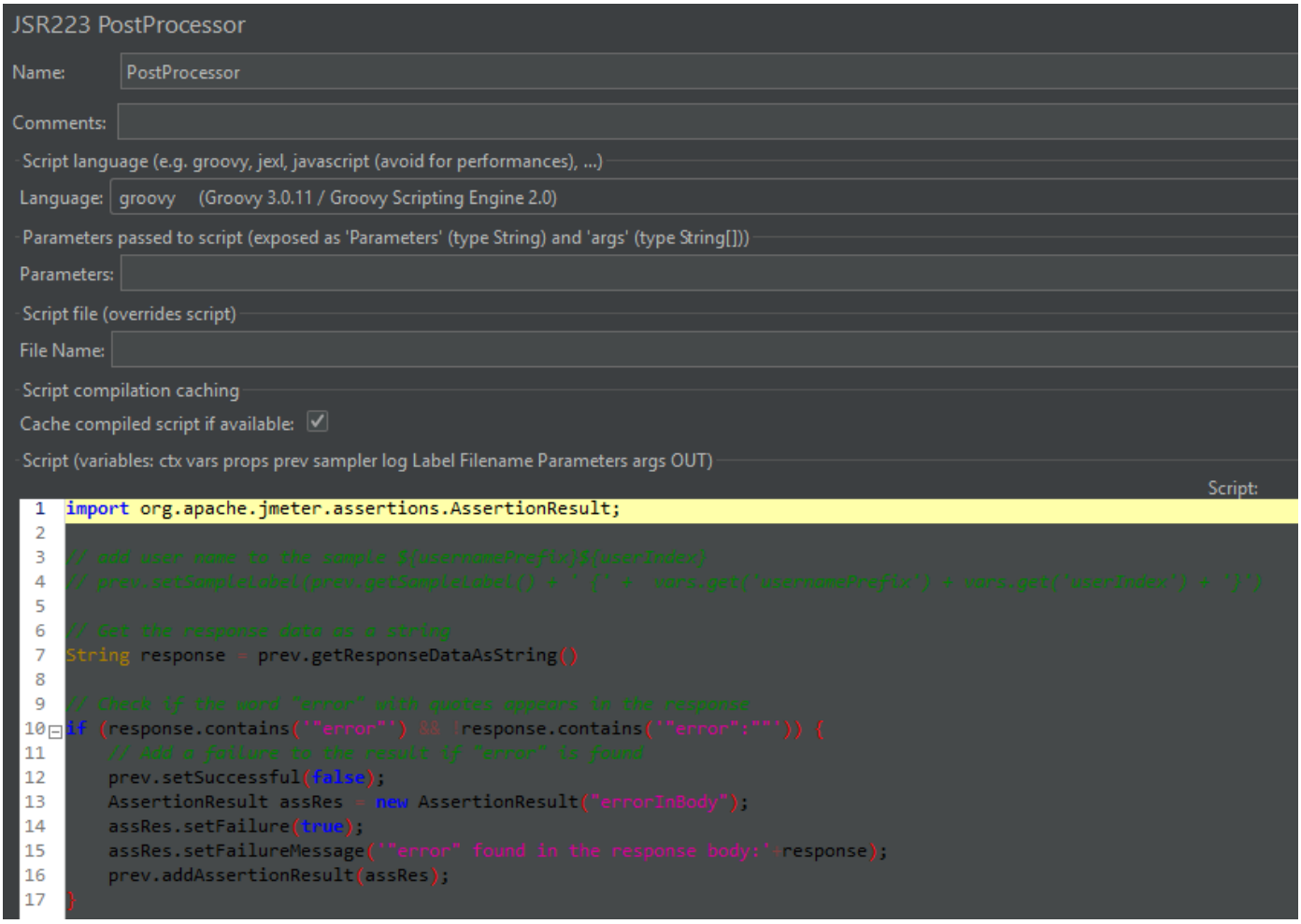
Fill the following:
Name: PostProcessor
Script:

```
import org.apache.jmeter.assertions.AssertionResult;

// add user name to the sample ${usernamePrefix}${userIndex}
// OPTIONAL – remove next line comment "//" to enable
// prev.setSampleLabel(prev.getSampleLabel() + ' {' +  vars.get('usernamePrefix') + vars.get('userIndex') + '}')

// Get the response data as a string
String response = prev.getResponseDataAsString()

// Check if the word "error" with quotes appears in the response
if (response.contains('"error"') && !response.contains('"error":"')) {
   // Add a failure to the result if "error" is found
   prev.setSuccessful(false);
   AssertionResult assRes = new AssertionResult("errorInBody");
   assRes.setFailure(true);
   assRes.setFailureMessage('"error" found in the response body: '+response);
   prev.addAssertionResult(assRes);
}
```



3.4. User Authentication

This is where we orchestrate the injection of different users for the various tests rather than reusing a single user repeatedly.

3.4.1. Authenticate user

Add a “HTTP Request” node [Right-click on **Thread Group**-> Add -> Sampler -> HTTP Request]

Fill the following[1]:

- Name: Authenticate user
- Protocol: ${schema}
- Server Name or IP: ${host}
- Port Number: ${port}
- HTTP Request: POST
- Path: /API3/authentication/authenticateUser
- Body Data:

```
{
  "username" : "${usernamePrefix}${userIndex}",
  "password" : "TestUserPassword[2]"
}
```

[1] Use Pyramid help to create the right API authentication requests for the different Authentication type.
[2] Replace "TestUserPassword" in the request body with the password of your users.

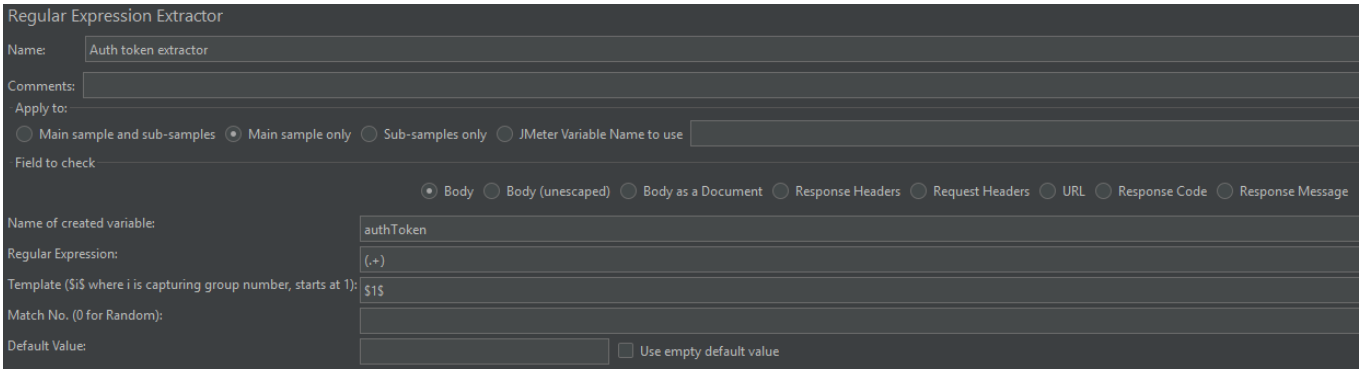
HTTP Request
Name:
authenticate user
Comments:
Basic
Advanced
Web Server
Protocol [http]:
${scheme}
Server Name or IP:
${host}
Port Number:
${port}
HTTP Request
POST
Path:
/API3/authentication/authenticateUser
Content encoding:
Redirect Automatically
Follow Redirects
Use KeepAlive
Use multipart/form-data
Browser-compatible headers
Parameters
Body Data
Files Upload
{
"username" "${usernamePrefix}${userIndex}",
"password" "snap3535!"
}

3.4.2. Auth token extractor

Add a “Regular Expression Extractor” node [Right-click on **Authenticate user** -> Add -> Post Processors -> Regular Expression Extractor]

Fill the following:

- Name: Auth token extractor
- Name of created variable: authToken
- Regular Expression: (.+)
- Template: 1



3.5. Recording Controller and Results

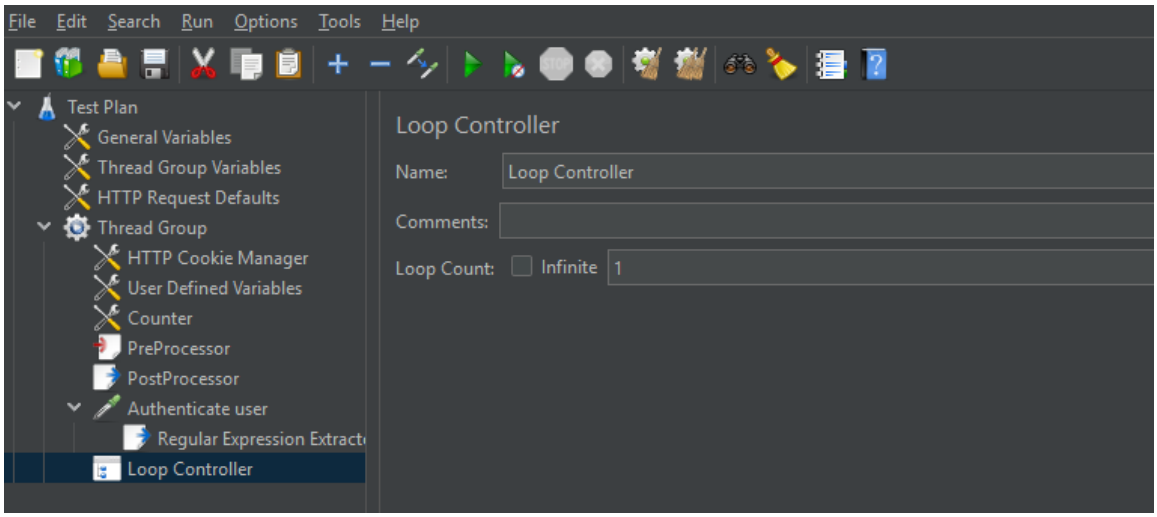
3.5.1. Loop Controller

Add a “Loop Controller” node [Right-click on **Thread Group** -> Add -> Logic Controller -> Loop Controller]

Fill the following:

- Loop Count: 1

Note: if a scenario needs to be run more than once, change this value to the wanted number of times each user should run the full scenario



3.5.2. Recording Controller

Add a “Recording Controller” node [Right-click on **Loop Controller** -> Add -> Logic Controller -> Recording Controller]

Note: this node will be filled in section 4 while recording the scenario, to clear this node for future scenario, the “Clear all the recorded samples” button can be clicked

3.5.3. View Results Tree

We add this node to collect the results of the tests and can be set to export the results as csv or xml. The results should then be analyzed to see test performance (see section 6 below).

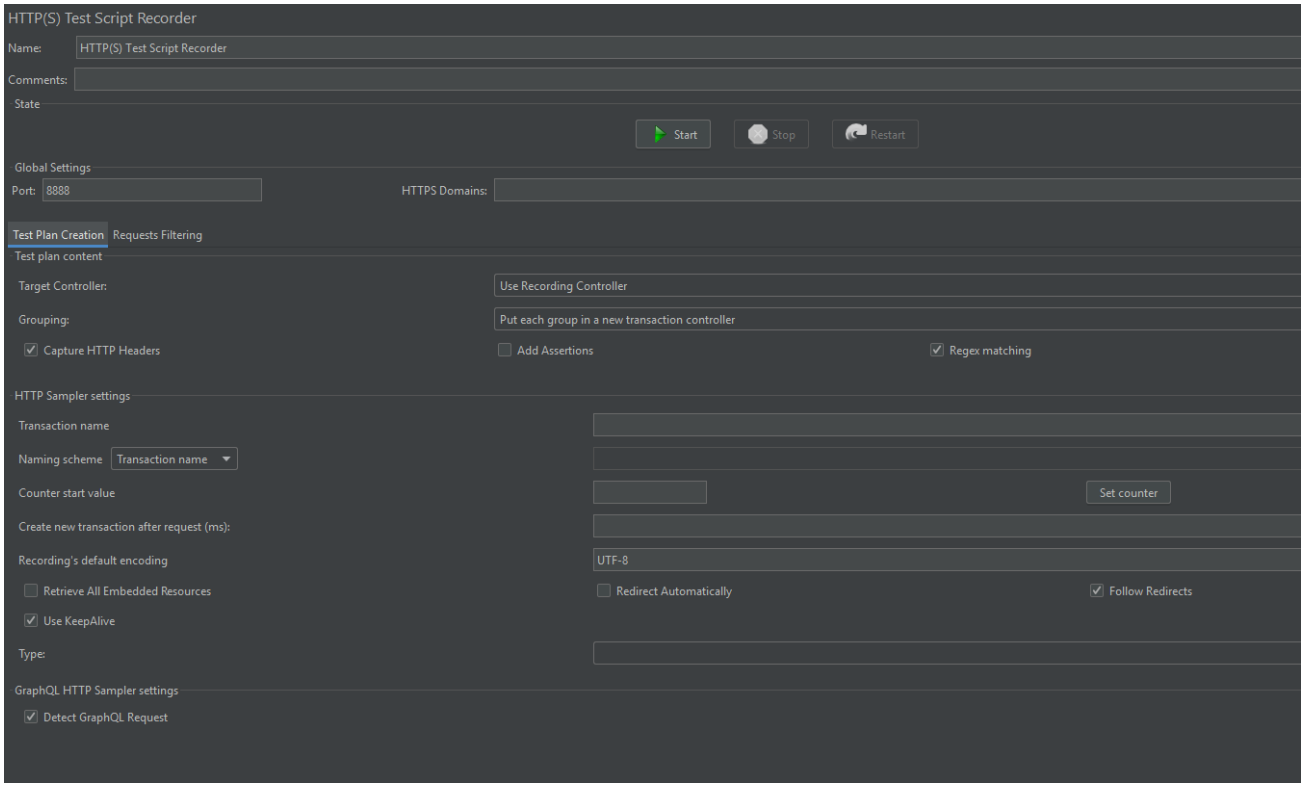
Add a “View Results Tree” node [Right-click on **Thread Group** -> Add -> Listener -> View Results Tree]

3.5.4. HTTP(S) Test Script Recorder

Add a "HTTP(S) Test Script Recorder" node [Right-click on **Test Plan** -> Add -> Non-Test Elements -> HTTP(S) Test Script Recorder]

Fill the following:

- Grouping: "Put each group in new transaction controller"
- Naming scheme: "Transaction name"
- Regex matching: check this checkbox

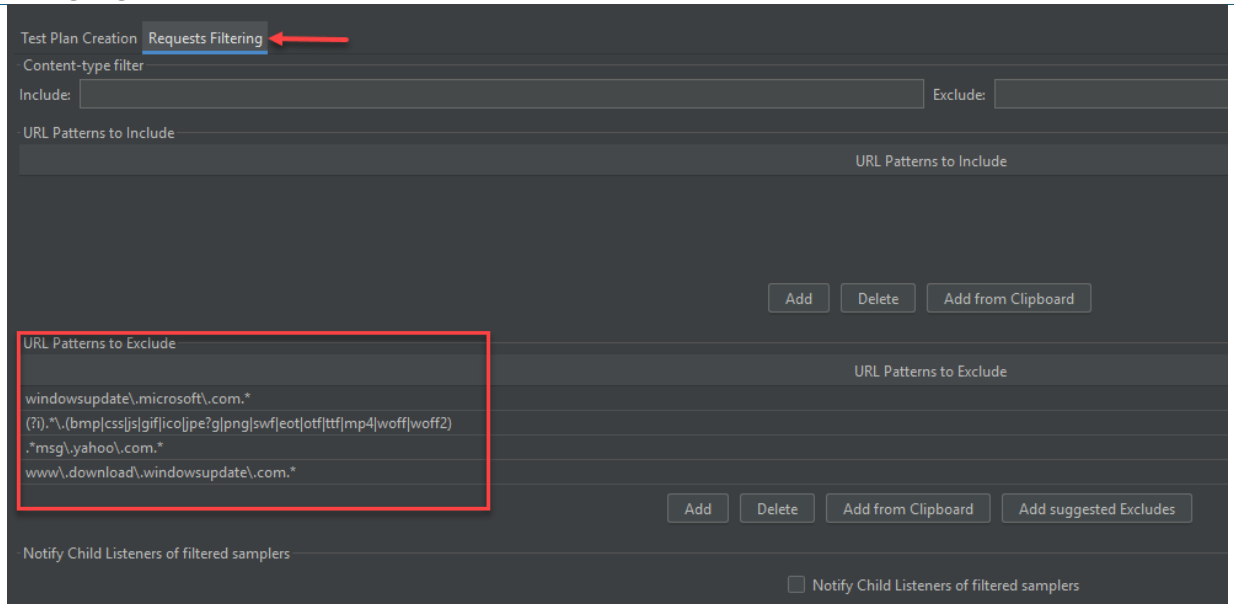


Go to "Requests Filtering" tab, and add the following to the URL Patterns to Exclude (these patterns should cover most non-relevant requests, but can be extended for specific use-cases):

Tip: This action can be completed by copying all rows and clicking "Add from Clipboard."

URL Patterns to Exclude
windowsupdate\.microsoft\.com.*
(?i).*\.(bmp|css|js|gif|ico|jpe?g|png|swf|eot|otf|ttf|mp4|woff|woff2)
.*msg\.yahoo\.com.*
www\.download\.windowsupdate\.com.*
toolbarqueries\.google\..*
http?://self-repair\.mozilla\.org.*
tiles.*\.mozilla\.com.*
.*detectportal\.firefox\.com.*
us\.update\.toolbar\.yahoo\.com.*
.*\.google\.com.*/safebrowsing/.*
api\.bing\.com.*
toolbar\.google\.com.*
.*yimg\.com.*
toolbar\.msn\.com.*
(?i).*\.(bmp|css|js|gif|ico|jpe?g|png|swf|eot|otf|ttf|mp4|woff|woff2)[\?;].*
toolbar\.avg\.com/.*

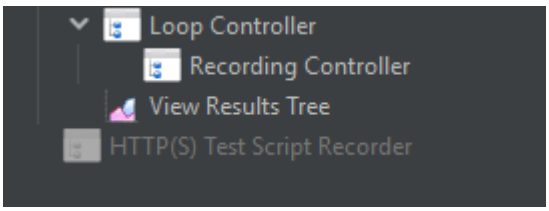
URL Patterns to Exclude
www\.google-analytics\.com.*
pgq\.yahoo\.com.*
safebrowsing.*\.google\.com.*
sqm\.microsoft\.com.*
g\.msn.*
clients.*\.google.*



Additional setting for HTTPS or certificate can be found here

After filling these details, right click on the node and select “disable”, this node is used manually and not as part of the test plan.

The node will become grayed-out.



4. Recording a Test with JMeter

4.1. Overview

This section will describe the process of planning (4.2) and recording(4.3, 4.4, 4.5) of a test scenario in JMeter that you then will deploy (later) with the test skeleton described above.

Section 4.2 contains suggestions and notes necessary for a true and reliable testing scenario.

Failing to follow these suggestions can cause tests to produce misleading results. It may also cause unexpected results on the platform causing the system to run out of memory, lock up threads and/or crash.

4.2. Planning the test

While planning a test a few key concepts should be noted.

4.2.1. Scenario test vs Operation test

Creating a test is typically used to check different user operations. It key to remember that a typical user does not login to the system to perform a single operation and then logout. Only to re-login and perform the next operation (and so on). A more real scenario is creation of a full flow of operations. This could include opening multiple content items, interacting with them using time gaps (like thinking time, described in section 4.4.1), before moving on to other content and so on.

Make sure that any flow begins from a logical start point. Some operations require previous operations to run first (e.g. running queries after a proper “Opening”). Skipping natural steps will create logical and structural errors and will affect operations on the engine.

4.2.2. Extreme operations

Some operations in the system may be “heavier” - for example first opening of a dashboard may cause running queries over slicers and over all visible graphs. While some user operations can use cache and partial queries which can be faster. Using a mix of these activities will give a more realistic view of how the platform will respond, rather than focusing on extremes (in either direction).

4.2.3. State changing operations

While “normal” users may use operation like save, move or import an item, these operations change the state of the system (future users need to have the new item location or content).

State changing operations are complicated and cannot be simulated easily via a standard stress test. Some operations cannot be executed more than once (e.g. deletion of item). In fact they will create logical errors rather than check system performance.

4.2.4. Test static data

Testing a dashboard built using data that may change may cause unexpected results. While this is a natural use of the platform, accommodating such changes in the stress test may be complicated. An example of this kind of error can be if a dashboard uses a slicer value based on the data and the selected value has been removed. In some cases it can cause an empty response – which may affect the results of the testing.

4.3. Test Recording Setup in JMeter

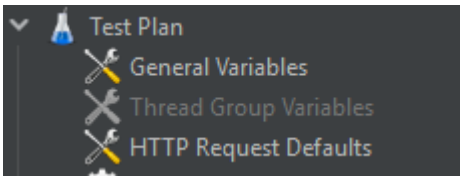
Before recording a test, the following steps should be taken.

4.3.1. Disable Thread Group variables node

JMeter replace all variables values in the requests with the variable to help the creation of tests, while this is mostly needed, some of the variables are relevant to the running of the test and not to the test itself (e.g. userCount or rampUp ....), to make sure that the JMeter will not alter these values in the recorder request, the runtime variables need to be disabled during the recording of the requests.

To disable the “Thread Group variables” node, right click on it and click disable.

Success should result in a grayed-out node.

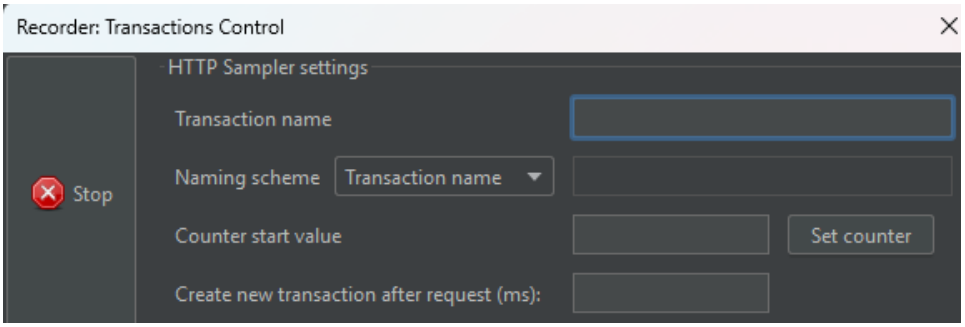


4.3.2. Proxy setup

Recording of network requests with JMeter works by redirecting network requests through proxy server created by the JMeter “HTTP(S) Test Script Recorder” node.

To run the proxy server, click on the “HTTP(S) Test Script Recorder” node, then click on the “Start” button.

This is how a successful Proxy server looks like:

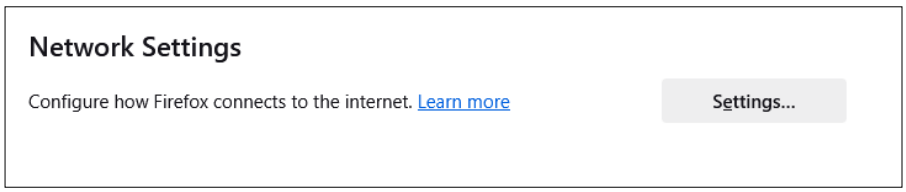


4.3.3. Connecting The proxy server to your machine

Connecting your browser to the proxy server is needed for recording, some browsers (e.g. google chrome) require the use of the OS proxy setting that may cause unwanted requests to be recorded.

While other browsers (e.g. Mozilla Firefox) that allows setting of the proxy server to the browser itself.

To set proxy on firefox go to settings -> General -> Network Settings -> Settings.

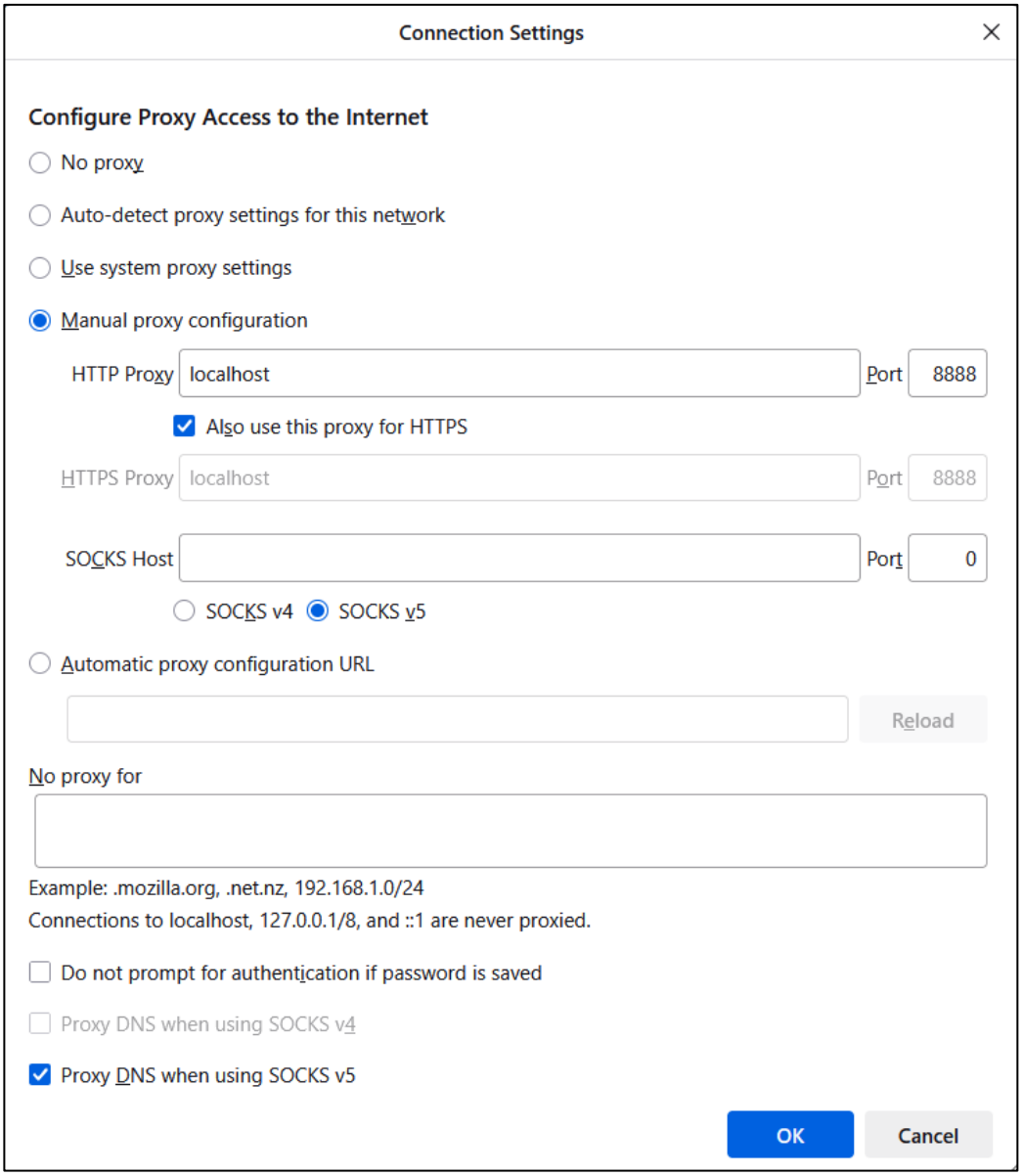


Select “Manual proxy configuration”.

And fill in the following:

- HTTP Proxy: localhost[3] Port: 8888
- Select “Also use this proxy for HTTPS”.

And save by clicking “OK”.

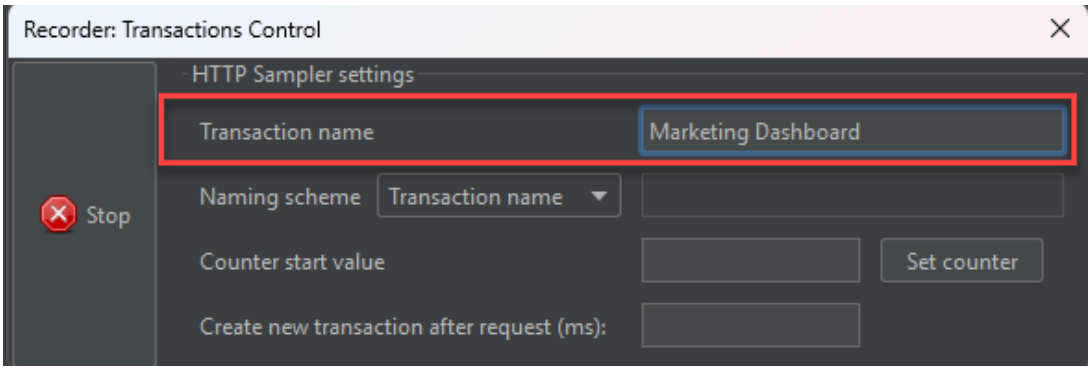


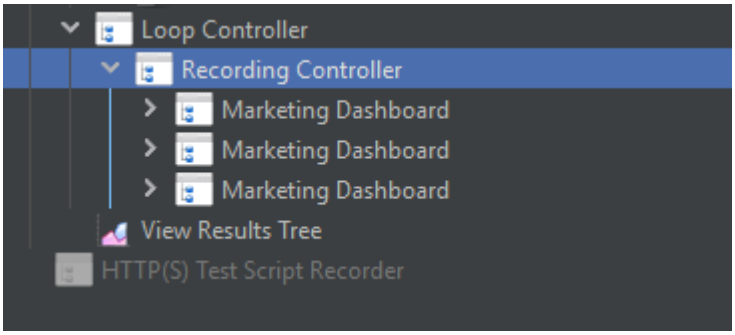
[3] “localhost” can be replaced with the hostname or IP of the machine running the JMeter proxy

4.4. Recording the test

After the proxy server is up (4.3.2) and connected to a browser (4.3.3), start performing the operations that will be tested.

- Each request group (request send in close time proximity, which can be configured) are recorded into a group inside of the “Recording Controller” node.
- Make sure to give sufficient time between operations like a real user would do (read some of the data before continuing and changing the next).
- While recording the name of each group can be set in the proxy popup. This name will appear for each group and request, this can help in case of a long scenario where you might want to mark each step with a different name.





4.4.1. Thinking Time Configuration

As described in section 4.2.1, a real user will think between operations while a code will run without gaps. So a wait gap between user operations is recommended for real world testing.

In addition, real users are not robots, and each user will take a different amount of time between operations.

Random thinking time also helps with non-realistic peaks (where 100 users click on the same button in the same moment).

To configure the default Thinking time, a change needs to be made to the JMeter configuration file that sits in "apache-jmeter-5.5\bin\jmeter.properties"

Add the following to the end of the file:

```
#---------------------------------------------------------------------------
# Think Time configuration
#---------------------------------------------------------------------------
# Default constant pause of Timer
think_time_creator.default_constant_pause=3000

# Default range pause of Timer
think_time_creator.default_range=500
```

- think_time_creator.default_constant_pause = Static thinking time between operations in millisecond
- think_time_creator.default_range = Randomness in the thinking time

Each Think-time node will wait "default_constant_pause" + [0 – "default_range"].

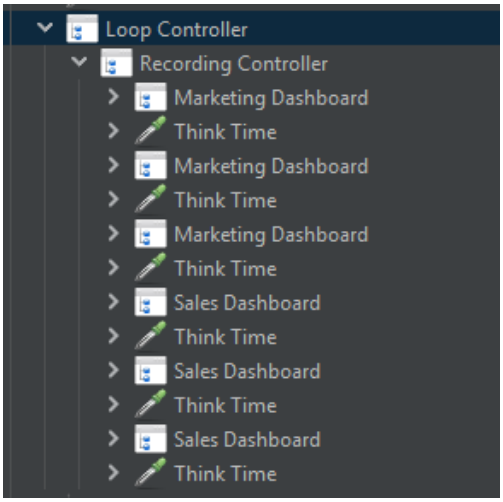
In the example above it will be between 3,000ms (3 sec) and 3,500 (3.5 sec).

After changing the configuration file, save and reload the JMeter program.

4.4.2. Adding Thinking Time to all recorded requests

Right-click on the "Recording Controller" node and click on "Add Think Times to children".

Now a think time node will be added after each Request group, which means that any "fast" request will run without delay, but between big operations a think time will be added.



4.5. Recording cleanup

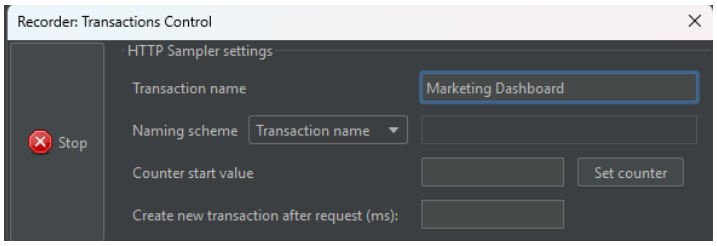
Once recorded, you'll likely want to cleanup the recording setup - reversing the operations (4.3.1 above).

4.5.1. Enable Thread Group variables node

Right click on the "Thread Group variables" node and click enable.

4.5.2. Stop Proxy server

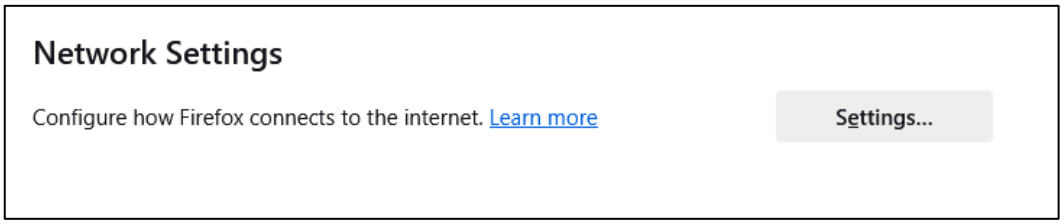
On the proxy popup, click on "stop".



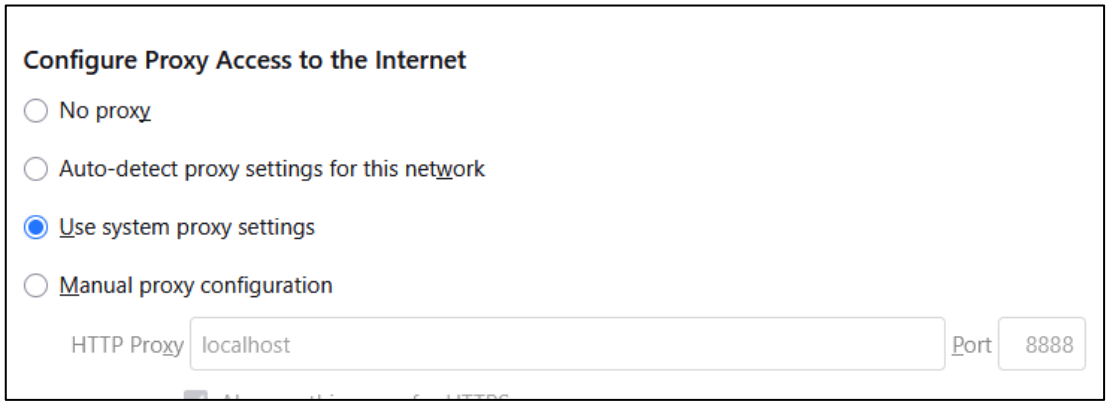
4.5.3. Disconnect Proxy server from browser

Reversing of operation described in section 4.3.3.

To set proxy on Firefox, go to settings -> General -> Network Settings -> Settings.



Select "Use system proxy settings".



5. Running JMeter Tests

Now, we are ready to run the tests in the testing skeleton.

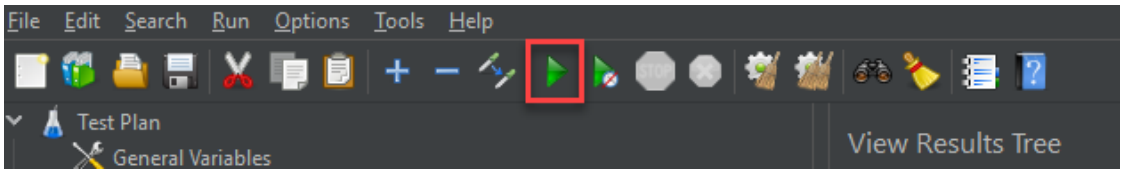
5.1. Configure test

Test parameters to configure:

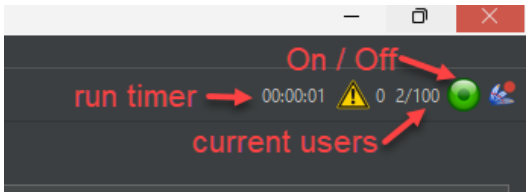
Ref	Parameter
3.2.2	User count
3.2.2	User ramp-up time
3.5.1	Number of iterations of the scenario

5.2. Running the test

To run the tests, go to the “View Result Tree” and click the green play button.



After starting in the right part of the screen will have a run timer and a counter that shows the current number of users, it will start from 0 and will scale up to “userCount” and it will take “rampUp” second to get from 0 to the number of users.



When the user finishes the scenario (defined by the recorded requests and the loop controller [3.5.1]) the counter will start to go down until all users are done and then the On/Off indicator will turn off.

6. Analyzing Test Results

After running a test session the results can be reviewed in various ways.

The raw results from the JMeter CSV contain each request that each user sent over the duration of the test with data of the request name(label), sent time (timestamp), duration of the request(elapsed), the sending user (threadName) and other field that can be used in specific scenarios.

This section will describe some suggestions for test results analysis. Test results are delivered as output generated based on the settings above (see 3.5.3).

6.1. User total scenario time

To calculate the total scenario time we need to subtract the last user request end time from the first user request start time:

Divide all requests by the thread name (threadName in the CSV). Each thread is a test user.

Those requests are all the requests sent by a singular user.

For all the requests of the user find the first time and last time using the timestamp field.

Calculate the time by:

$$Total_time = (Last_request_timestamp + Last_request_elapsed) - First_request_timestamp$$

This value can be checked in different ways:

- **min/max** – Check the difference between the “slowest” user to the “fastest” user to understand whether the difference is in normal range. Normal range can differ depending on the scenario length and its composition and includes the randomness of the thinking time.
- **First/last** – Due to ramp up period (3.2.2), the first user starts with no load at all, but finishes with the most load (all of the users are already running). The last user is the opposite, starting when all of the users are running and finishes alone. The time difference between those should be low and helps to understand how the change of the load over time affect the time. Changing the ramp up time can help to create a constant load instead to creating load peaks.
- **Average for varying number of users** – Check the average user time for single user and compare it under load. The time can be different, but when there is no throttling (resources or application queue sizes) it should be close to the single user time.

Keep In mind that due to the thinking time aspect(4.2.1,4.4.1) each variation of the thinking time add randomness to all time calculation in the results. If each thinking time node adds randomness of 500ms, than the scenario time of slowest and fastest user in the test can differ by 0.5 * number of user operations, and for large tests this can sum up to substantial time difference.

6.2. Sub-Scenario time

Just like the previous section, the time for each sub-scenario can be obtained and reviewed. A sub-scenario is a group of user actions that have been recorded under the same name.

To calculate this time use the same as the total time just divide all requests by the thread name and label(

Depends on the settings in 3.5.4). The label will be the “Transaction name” that filled in 4.4.