Technical Assessment – Performance Test Strategy

1. Objectives:

The key objective of the technical assessment was to design performance test for the Restful Booker API endpoints ensuring full coverage of APIs. This included designing robust Jmeter script which is modular with re-usable components and easy to maintain.

2. Testing Approach / Strategy

To achieve the defined objectives, the approach was categorised into:

- Defining scope
- Understand the requests and responses for each API
- Understand the request headers to avoid failures
- Functionally testing each API and its variants
- Define user journeys that reflect real world usage
- Design Jmeter script that is modular, easy to maintain, provide full coverage of API's and their variations
- Include different performance test types – Load, Soak, Spike and Stress test
- Analysis and Reporting capabilities

3. Scope

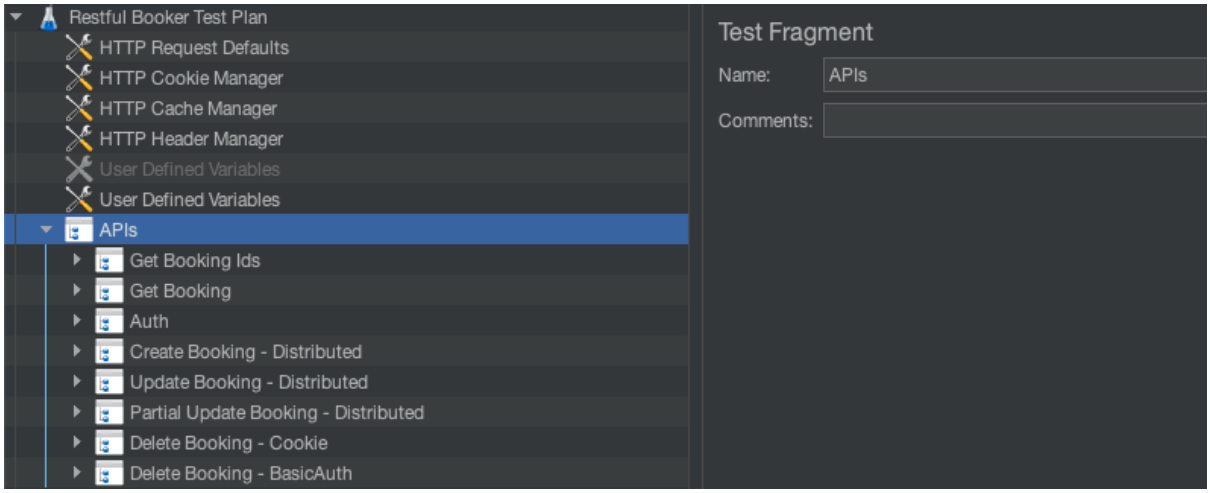
Based on the analysis of the Restful Booker API documents, the following endpoints were part of the test scope:

Name	Endpoint	Method
Auth - CreateToken	/auth	POST
Get Booking Ids (All IDs)	/booking	GET
Get Booking Information	/booking/{id}	GET
Create Booking (JSON,XML and URLEncoded)	/booking	POST
Update Booking (JSON,XML and URLEncoded)	/booking	PUT
Partial Update (JSON,XML and URLEncoded)	/booking	PATCH
Delete Booking (Cookie and AUTH)	/booking	DELETE

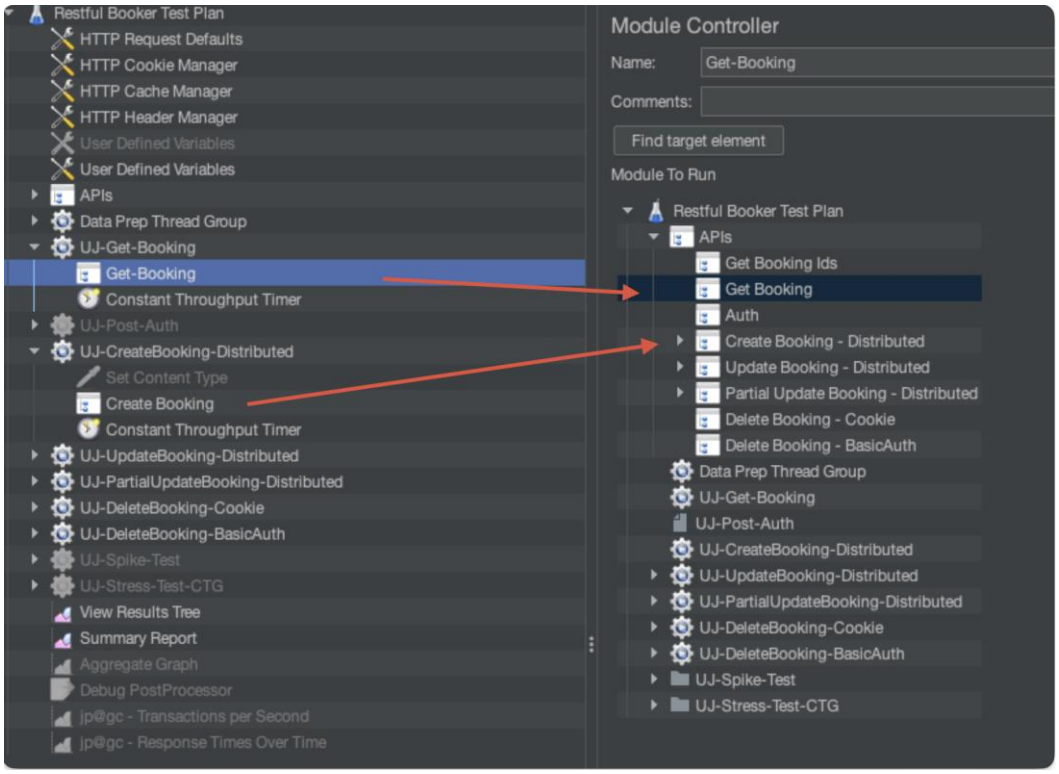
4. Jmeter Script

Jmeter script was designed to be modular, flexible at the same time robust to provide full coverage. Special attention was given to make the script less complex and easier to maintain for future requirements.

The below screenshot shows that all the API's were defined under "Test Fragment" and within that each API is defined within "Simple Controller".



The Thread Group makes use of "Module Controller" to point to the correct test api fragment/simple controller. The main benefit of modularising this way is the ease of adding new user journeys or modifying existing user journeys.



The script is driven by use of a properties file. This file has information required for scripts to run tests at required levels. The file in this assessment used is “test.properties” and it looks like below:

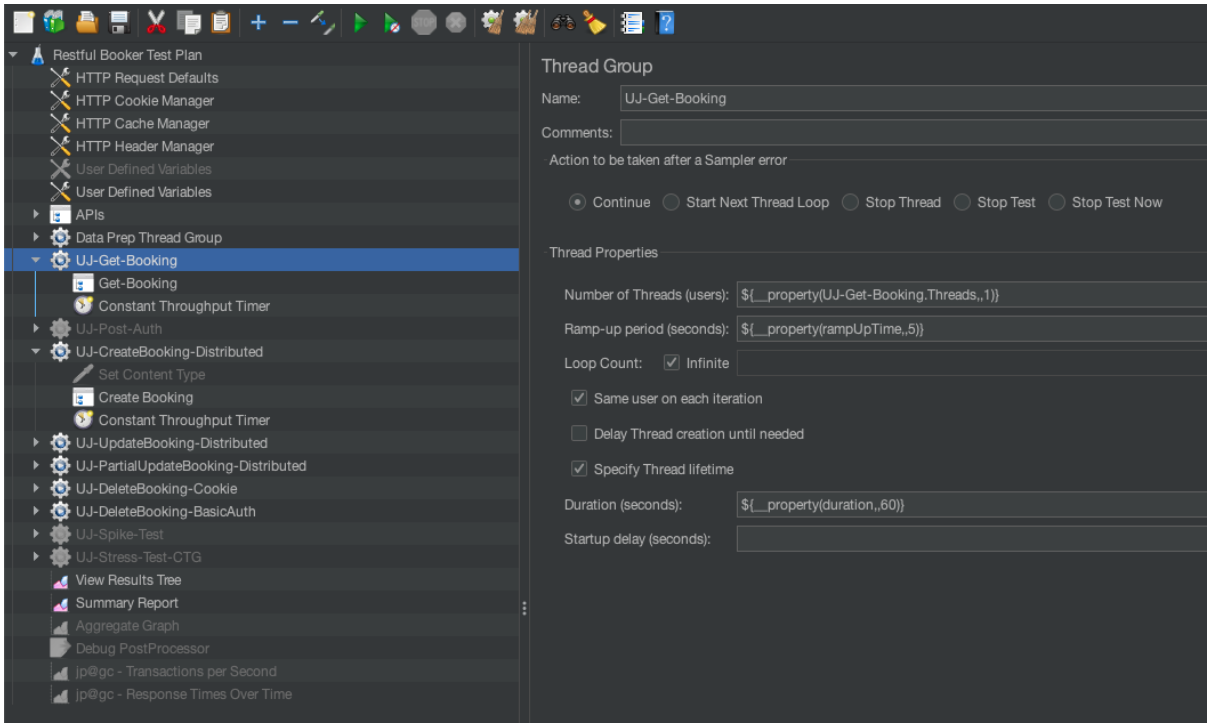
```
#########################################################
# Test parameters

# Test Duration in seconds (5 minutes=300)

duration=300

# Load Test for 1 Hour
# duration=5400

# Soak Test for 24 Hours
# duration=84600

# Ramp-up Time in seconds (5 minutes=300)
rampUpTime=30

#########################################################


# Load Test Profile - Based on Requests per Minute (uses Constant Throughput Timer) per Request Type

UJ-Get-Booking.Threads=10
UJ-Get-Booking.rpm=100

UJ-CreateBooking-Distributed.Threads=10
UJ-CreateBooking-Distributed.rpm=60

UJ-UpdateBooking-Distributed.Threads=5
UJ-UpdateBooking-Distributed.rpm=40

UJ-PartialUpdateBooking-Distributed.Threads=3
UJ-PartialUpdateBooking-Distributed.rpm=20

UJ-DeleteBooking-Cookie.Threads=2
UJ-DeleteBooking-Cookie.rpm=20

UJ-DeleteBooking.Threads=2
UJ-DeleteBooking.rpm=20

UJ-DeleteBooking-BasicAuth.Threads=5
UJ-DeleteBooking-BasicAuth.rpm=30

UJ-DeleteBooking-Cookie.Threads=5
UJ-DeleteBooking-Cookie.rpm=30
```

This method makes it much simpler to handle tests with different loads. Rather than changing the Jmeter script, it makes it easy to just modify the properties file. Another benefit is to have multiple properties file based on test requirements – load, soak, stress etc.

The script can be executed as below:

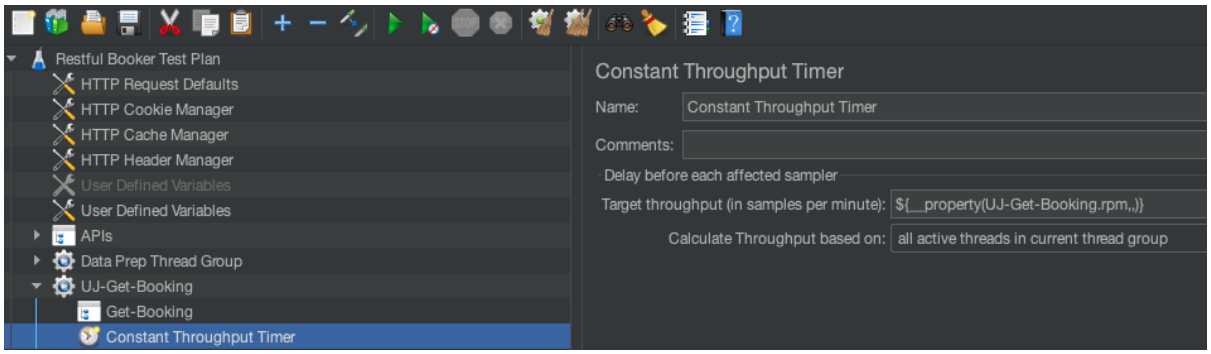
$> jmeter.sh -n –t <path>/<script_name> -p <path>/<properties_file_name>

The Thread Group details are as shown below. They are read from the properties file at the start of test execution.

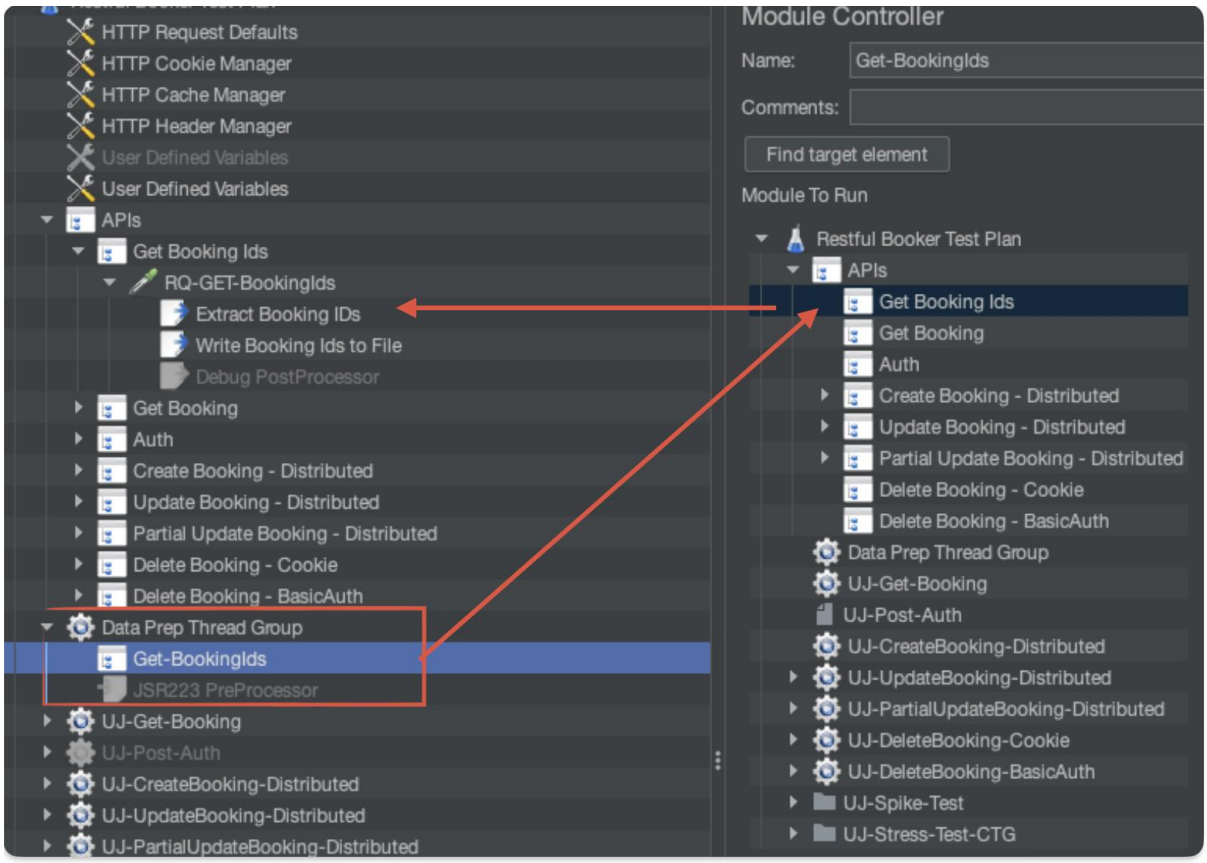


The above example is based on assumption that the requests per seconds/minutes are available from the workload model which can then be fed into the properties file. This implementation helps keep the Jmeter script not subject to un-necessary changes for different test types which can be driven by a simple text properties file.

For each thread group to achieve the required throughput, the script uses “Constant Throughput Timer”. The value for this should be derived from the workload model.

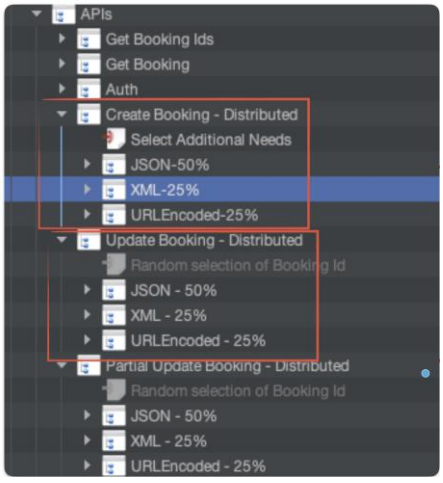


The script uses “SetUp Thread Group” which gets executed once per test run. This is used to setup data for other user journeys. In this example, the “Setup Thread Group” runs the GET /booking to extract all booking ids and store them in an external file. This file is then used by other user journeys like Update Booking.



To make the script generic, it has used multiple correlation post processors like “JSON Path Extractor”, “Regex Extractor” and “Boundary Extractor”. It also makes use of JSR Processors to write data to external files and process responses.

The script also makes use of “Throughput Controller” to blend the different requests under one thread group. This is particularly useful for creation of Stress and Spike scenarios. It is also used for Update Booking API’s which has different requests types like JSON, XML and URLEncoded.



5. Test Execution

The current test script supports different performance test types. The tests with high load levels are recommended to be run in non-gui mode:

$> nohup jmeter.sh -n –t <Script.jmx> -p <Test.properties> -l <Results.jtl> &

Load Test

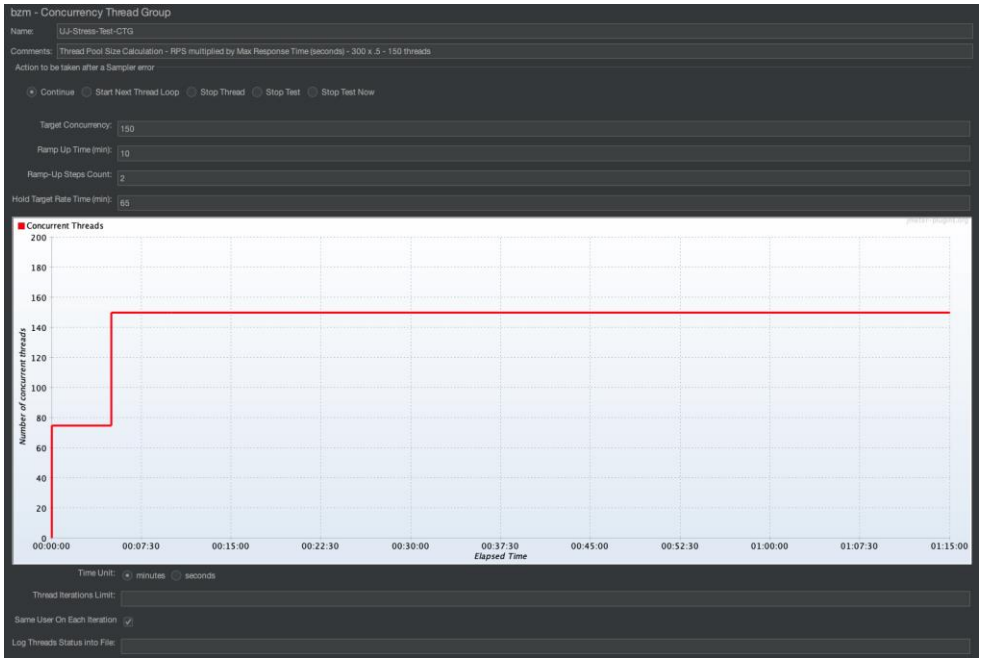
As the test is driven by the properties file, the only details that need to be fed into the file are test duration, rampup and user journey details (Threads and RPM).

Soak/Endurance Test

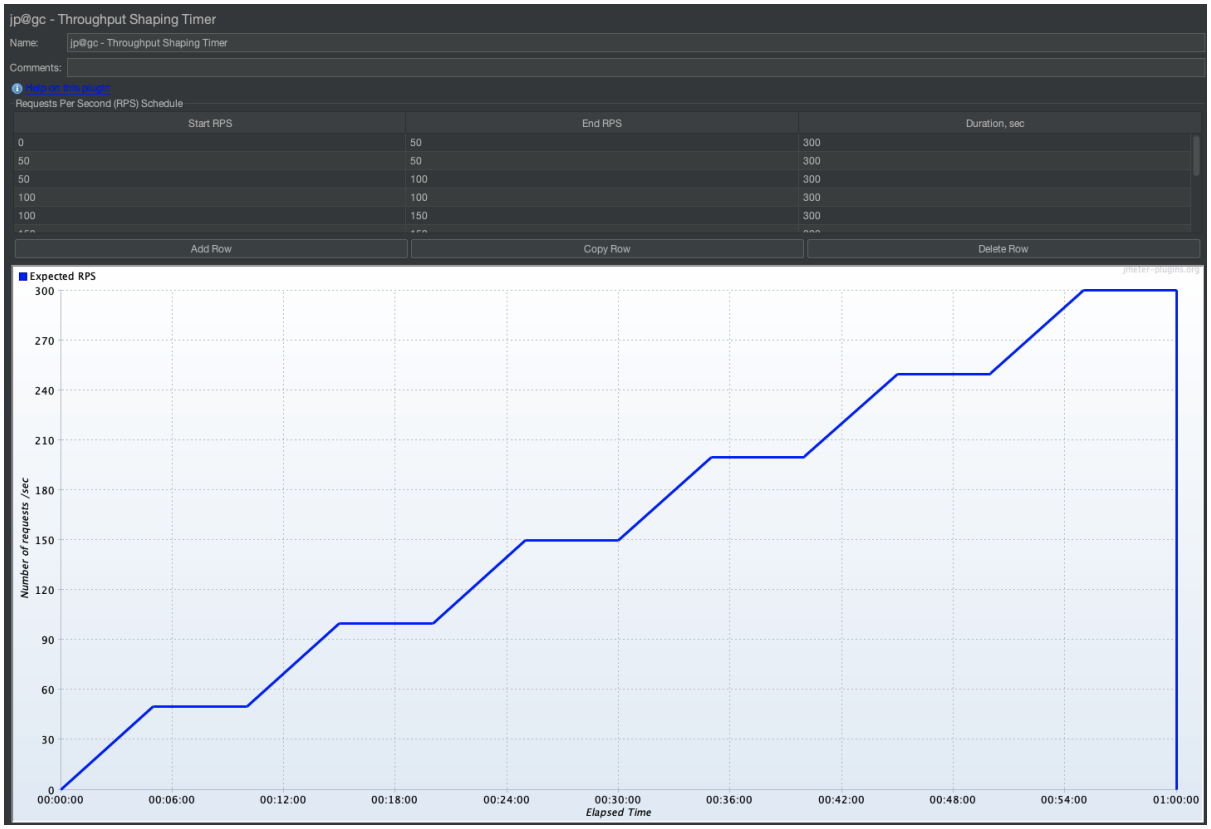
Assuming that the same load test profile is used for soak test, the only change to properties file required is the change to “duration” field set to something really high e.g. for 24 hour soak test, set “duration=84600”

Stress Test

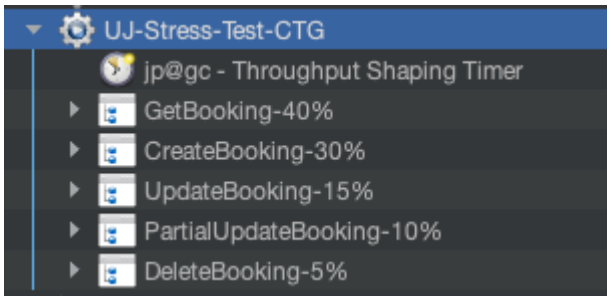
The key requirement for stress test is to find the capacity or breaking point, it is best to drive stress test based on requests. The current test uses Blazemeter “Throughput Shaping Timer” alongwith “Concurrency Thread Group” to ramp-up the load in stages.



The ramp-up stages are defined to start with 50 RPS and then run at steady state before ramping up again.

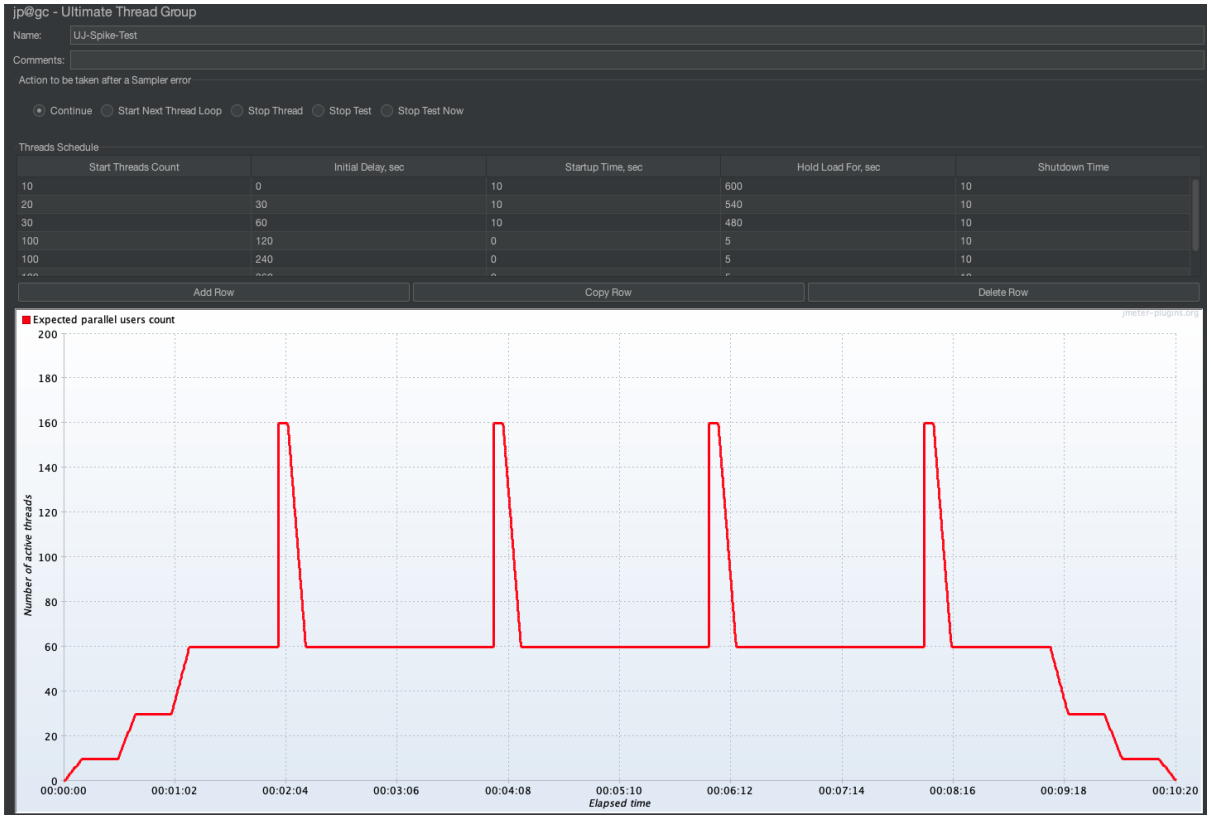


This thread group executes all the APIs based on percentages which can be determined from workload model or other sources.



Spike Test

For the spike test, the script uses “Ultimate Thread Group” which is based on threads ramping up. This allows spikes in requests to be introduced during different stages of the test.



6. Analysis and Reporting

Jmeter tests are best run in non-gui mode and output the test metrics in the form of jtl files.

$> jmeter –n –t <Script> -p <Properties> -l <JTL File>

Also make sure that the Jmeter has been provisioned with enough heap size to avoid it becoming the bottleneck. If using Limux based load generators, uplift the kernel level metrics like number of open files, ephemeral local port range, tcp timeouts and process limits. This is quite critical when running large load tests.

For metrics collection, it is recommended to use combination of Grafana/InfluxDB to send metrics to. This allows analysis and comparisons between runs and have historical data. There is an already available listener in Jmeter.

Apart from this, have the observability platform by using APM tools like Dynatrace, New Relic, Datadog, Kibana/Elastic Search, Grafana and Prometheus to monitor infrastructure and application metrics, logs and traces for the test environments to identify performance issues and bottlenecks.

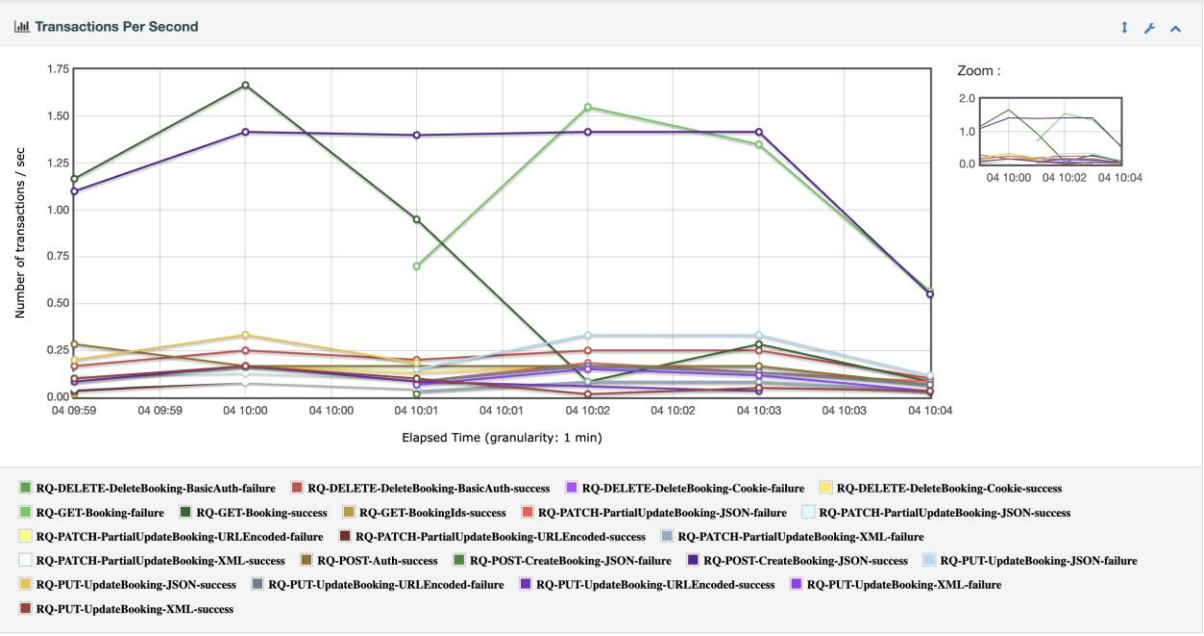
The key metrics to look are:

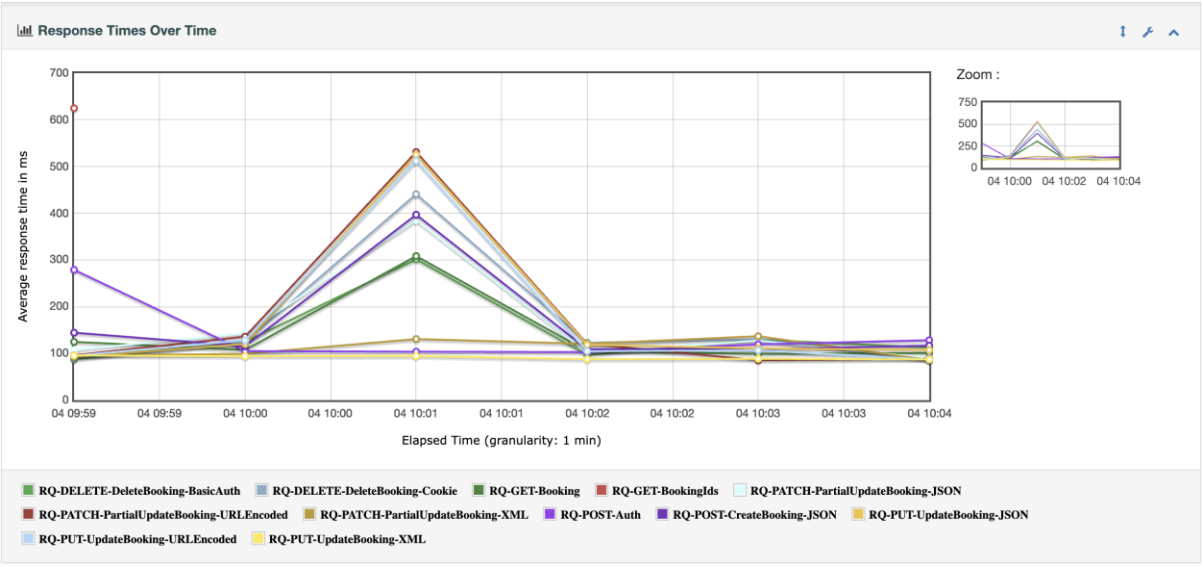
- CPU Vs Response Times

- Error Rate
- CPU/Memory/Disk IO
- If Java based app then JVM metrics, Garbage Collection, GC Pause times, Connection Pools usage and Thread usage
- For DB, it can be Top 10 SQLs

Jmeter allows creation Dashboard Reports using JTL files from the test. It provides useful information like Throughput, Response Times, Errors, Latency etc.

Some of the graphs from the test executed:





7. Test Execution – Technical Assessment

A load test with all the user journeys was executed as part of the assessment. It executed the following journeys:

User Journeys	Threads	RPM	
UJ-CreateBooking-Distributed	**10**	**60**	
UJ-DeleteBooking-BasicAuth	**5**	**30**	
UJ-DeleteBooking-Cookie	**2**	**20**	
UJ-Get-Booking	**10**	**100**	
UJ-PartialUpdateBooking-Distributed	**3**	**20**	
UJ-UpdateBooking-Distributed	**5**	**40**	

The load was kept low assuming that the target application was not scaled for load tests. Even under this load, there were some failures observed especially around POST and UPDATE/DELETE calls returning HTTP 5xx errors under full load. It appeared that the application layer and/or database behind the application were unable to cope with the load which is expected due to scaling constraints.