Tero Suominen

PERFORMANCE TESTING REST APIS

Information Technology

2017

Satakunnan ammattikorkeakoulu

Satakunta University of Applied Sciences

PERFORMANCE TESTING REST APIS

Suominen, Tero
Satakunta University of Applied Sciences
Degree Programme in Computer Science
December 2017
Number of pages: 39
Appendices: 0

Keywords: REST, performance testing, automation

The subject for this thesis was performance testing REST APIs that had been implemented into a Java application. The purpose of this research was to come up with a method on how the performance and functionality of the REST APIs could be measured and tested within Profit Software.

The research consisted of two parts. First, I searched to find an existing software capable of being used for testing REST APIs. After selecting the tool that would be used to create the performance tests, a local test environment was set up that allowed us to estimate the capability of the software and the method of testing itself. The local environment consisted of the same components and software that could be used also in the already existing test environments within the company. This way moving the tests from the local environment into the actual test environment went smoothly.

With the help of this research we were able to detect issues with the functionality of some APIs, when they were under load. We were able to fix these issues in the implementation during the development phase and after changing the implementation we could verify that the APIs functioned correctly by using these same tests.

REST RAJAPINTOJEN SUORITUSKYKYTESTAUS

Suominen, Tero
Satakunnan ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Joulukuu 2017
Sivumäärä: 39
Liitteitä: 0

Asiasanat: REST, suorituskykytestaus, automaatio

Opinnäytetyön aiheena oli Java-sovellukseen toteutettujen REST rajapintojen performanssitestaus. Tutkimuksen tarkoitus oli luoda testausmenelmä jolla toimeksiantajayritys Profit Software voisi tulevaisuudessa varmistaa uusien rajapintojen suorituskyvyn ja toimivuuden rasituksen alla.

Tutkimus koostuu kahdesta osasta. Ensin etsittiin ja tutustuttiin sovelluksiin, joilla testausta pystyisi suorittamaan. Työkalun valinnan jälkeen rakennettiin paikallinen testiympäristö, jonka avulla pystyttiin arvioimaan valitun työkalun ja testausmetodin kelpoisuutta. Paikallinen testiympäristö koostui samoista komponenteista, mitä voitaisiin myös käyttää yrityksen jo olemassa olevissa testiympäristöissä. Täten testien siirtäminen lokaalista ympäristöstä varsinaiseen testiympäristöön sujui vaivattomasti.

Työn avulla pystyttiin havaitsemaan ongelmia joidenkin rajapintojen toiminnallisuudessa, kun niihin kohdistettiin enemmän rasitusta. Nämä ongelmat pystyttiin korjaamaan toteutuksesta vielä kehitysvaiheessa ja muutosten jälkeen rajapintojen oikeanlainen toiminta voitiin verifioida samojen testien avulla.

TABLE OF CONTENTS

DEFINITIONS

REST - Representational State Transfer

API - Application Programming Interface

PLP - Profit Life & Pension

HTTP - Hypertext Transfer Protocol

GUI - Graphical User Interface

CI - Continuous Integration

CSV - Comma Separated Values

URI - Uniform Resource Identifier

1 INTRODUCTION

As web and mobile applications are developing more and more to support end users fetching data directly from servers the need for creating interfaces for old software to support and handle such services has come very topical in many companies. For some systems that previously were used internally by companies this means that now their customers could see and manage their information directly. This makes the performance of the APIs get more and more crucial as the number of users and the performance load on the software and server increases.

This thesis focuses on testing the performance and reliability of REST APIs (Representational state transfer Application programming interfaces) implemented in Java application. Thesis was carried out in collaboration with Profit Software and the subject for this research and thesis came from the company.

Profit Software is a software company that offers solutions for insurance business and has currently approximately 230 employees in six different locations. The company headquarter is located in Espoo and side offices are in Tampere, Pori, Lahti, Tallinn and Stockholm. Since the foundation in 1992 Profit Software has served more than 40 clients in nine countries and are currently the industry leader in Finland and in the Baltic states with over 50% market share. Along with consulting services, the main software product Profit Life & Pension (PLP) is a web-based solution that covers all sales, care and claims processes needed for managing life, pension and personal risk products. (Profit Software 2017)

In order to succeed in this there was a study to look for a tool to use and a small demo to test and see how the actual setup works. Most of the tools were discovered online via blog posts related to Web API testing. For the scope of writing this thesis the focus was selected to be on couple of options and the actual creating of the performance tests.

In chapter two of this thesis the fundamentals and the functionality of REST APIs in a Java application is explained along with good practices what comes to performance testing. The scope of this thesis is further explained in the chapter three. In chapter four two most promising tools were compared and chapter five goes over creating performance tests with the selected tool in detail. Finally, in chapter six the results and outcome of this thesis is analyzed.

2 THEORY

2.1 REST Architecture

REST is an architectural style for designing networked applications. It created by Roy Fielding and introduced in his PhD dissertation where he defined the core set of principles, properties and constraints describing the architectural style of the World Wide Web. (Fielding [1] 2000).

These constraints allow distributed systems and applications to communicate and it guarantees the compatibility between different resources. Typical REST applications use standard HTTP methods (get, post, put, delete), as well as standard respond codes. They are used as they were originally intended, get for requesting information, post for creating, put for updating and delete for removing resources. In these kind of applications, the resources are addressed by their URIs.

REST comes from Representational State Transfer. "Representational" refers to the way a resource is represented, it can be anything that can be identified by a URI. "State" refers to the state of a resource, a resource has a state at the server and a client also has its own state. "Transfer" is used to manipulate and fetch data. Client can transfer states to the server in order to update the state of a resource on the server or it can get states from the server to update its own state.

While in most cases the protocol used is HTTP, other protocols that fit the constraints can be applied. It needs to support client/server-model, it needs to be stateless so all information is transferred within the messages, it needs to be cacheable so that multiple same requests return the same result and layered so that changes in the intermediaries do not affect the interfaces. (Fielding [2] 2007).

2.2 REST APIs in Profit Life & Pension

The REST APIs that will be covered in this thesis were created as customization on top of Profit Life & Pension-product within a customer project. The REST API layer is built to support online web-services so that insureds can interact with their policies directly online. Currently those use EJB-based Facade implementation that is now being replaced by these REST APIs. In the future the created REST APIs are planned to be included into the product itself.

The separate service API implementation for REST services allows PLP to continue to work like before, managing the policies and executing batches, etc. The original functionality isn't affected or changed, only now there is a new way for end users to interact with PLP through whatever client the customer uses. Service APIs also takes care of packing and receiving requests and responses. It separates the REST APIs from the rest of PLPs APIs and doesn't expose what is happening below this layer. All exceptions and errors are wrapped to a ServiceResult objects that are returned in a service response. This way the client used for sending the requests can get feedback of the status of sent requests, without exposing the inner workings of PLP.

3 SCOPE OF THIS THESIS

3.1 Purpose

The subject for this thesis came from Profit Software. The purpose was to research and develop a process that could be used to measure and test the performance of REST APIs implemented in the product. The goal was to find a suitable testing tool and method for testing the existing and future REST API implementations, which could be taken into use for all projects within Profit Software.
Testing the new REST implementations is important and needs to be done during development.

Long response times and possible issues that may be caused by the REST APIs can hinder the customers business and once the release goes into the production it is nearly impossible to debug and evaluate the cause for the issues. On the production side the server logging is minimal and without the sufficient data, debugging these kinds of issues and finding the root causing them is tedious and takes time.

Earlier different methods and tools have been used to test the functionality and the performance of REST-requests, but there has not been a unified way of creating and executing REST API performance tests. A couple of articles mentioning different tools and methods can be found in the company intranet, but there were no regularly performed tests in any project. For one customer project there were some tests created for few of the more important REST-requests in pure Java, using the Runnable-interface. At runtime it generated threads that sent multiple requests simultaneously directed to the same API. With these tests developers had been able to find and replace some thread-safety issues that came up during parallel execution.

I also took opinions of other employees into consideration of my research on how the testing should be done. Because of these opinions, I rather quickly abandoned the idea of creating my own test application or writing the tests by hand all together. The reasons for this were that it would have been very resource heavy in the future to maintain in the future. Also, creating it would have been very time consuming and it wouldn't have brought enough benefits or features that already existing tools couldn't provide.

The primary goal of the thesis was to research a method or a tool that could be used by developers to test new REST API implementations or verifying the functionality and performance after a change in implementation. It was important for the test execution to be able to find issues that were caused by having multiple requests sent in parallel and to catch any drastic declines of performance.

Because of the advice given by my co-workers I also looked into the infrastructure of various projects within the company and investigated the possibility of integrating REST API performance testing as a part into the already existing test automation-process. Within Profit Software there are multiple projects that have their own Jenkins CI-environments, which perform automated tests against builds deployed in the cloud. After being integrated into the automation the performance tests could be executed often and always against the latest build, making it faster to find decreases in performance and functionality issues caused by a recent change.

3.2 Research methods

First, I started looking into a REST API implementation developed as a part of one customer project and how it functioned within the product. With these APIs I started to go through different tools and methods on how the REST API testing could be done. For the testing I used the beta version of PLP, which was deployed into a local docker container as accustomed within Profit.

Articles and blog posts found from the internet were used to find about different tools and methods on how to test REST APIs. Some tools were already mentioned within the company intranet and I also heard about some test setups and tools used in other companies from my co-workers. For my evaluation of the various tools I created tests for the REST APIs and executed them locally on my machine.

After coming up with a proper tool and method for testing REST APIs we wanted to find out how the tests would be executed and how they could be integrated into a project infrastructure. As part of my research the selected tool and method for REST API testing were tested in one customer project within Profit Software. For these tests a local test environment that had all the components running was created and it was used to test and demonstrate the functionality of the REST performance tests. It was also used to show how the performance tests could be integrated as a part of the projects infrastructure and already existing test automation and to evaluate how it would suit for other projects as well.

4 RESEARCH AND IMPLEMENTATION

4.1 Tool evaluation

There were certain requirements that were taken into consideration when selecting the tool for testing. It needs to be able to send multiple requests in parallel, receive the incoming responses and validate the eligibility of the responses. Also for reporting and comparing it needs to measure and save certain values of the responses, for example the most important ones are response code, response message and response time. These could be used to compare the success rate of sent requests and measure the performance of the API.

There aren't many tools designed and created just for REST API testing, but almost every tool created for the purpose of testing a Web Service can be used for REST-testing. All tools that can send parallel requests and support HTTP-protocol can be used. Some of the tools primarily created for Web Service-testing have tools for recording user actions. These kinds of features don't bring any additional value for REST API testing. For REST API tests we can focus our simultaneous sent requests on a single URI, creating more direct load against that particular API endpoint.

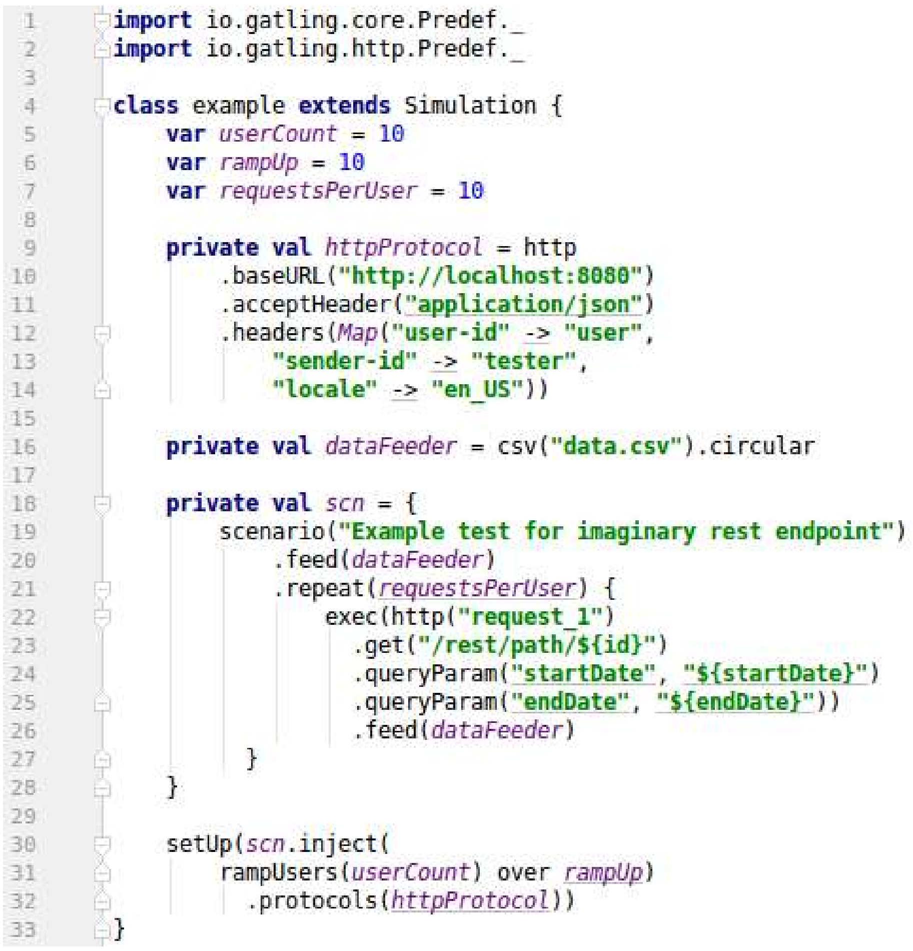
The calls must also be able to be parameterized. The resource might require certain parameters included with the request. During testing it is important that the REST-requests are distinct, in order to avoid cached responses from the product. Cached responses would make the test results distorted and incorrect. I also valued platform independence, in order to create tests that could be executed on multiple different environments and platforms.

During this phase I went through and became familiar with multiple programs used for load testing, some open source and some requiring a license or having a trial period such as SoapUI/LoadUI, LoadRunner, Grinder, Tsung, Wrk and Loader.io. I created very basic test suites with the help of their own tutorials or other internet sources. After I had a general idea and picture of the tools that are available and what can be done with them I started to focus more on two of the most promising candidates, Gatling and Apache JMeter. Both of them are open source applications, they support external data sources for parameterizing requests and can be rather easily integrated into test automation with Jenkins. Some of the previously mentioned tools also support these features, but I found these two to be more straight forward, license free and sufficient for testing REST APIs.

4.1.1 Gatling

Gatling is a tool created for load testing. It takes advantage of its asynchronous architecture, which allows it to implement virtual users as messages instead of dedicated threads, making running numerous simultaneous virtual users less resource heavy than other solutions. (Gatling [1] 2017.)

Gatling tests are called simulations. They are written in Scala-files and each Scala class represents its own load simulation. We can assign properties such as the URL, parameters and headers that are needed for the HTTP requests into variables using Gatling HTTP protocol builder. The testing scenario itself is also assigned into a variable. The scenario represents actions and requests sent by one user. The setUp()-method is used to actually define the execution of the test: which scenario is to be executed, what is the protocol and properties for it and how many users are created for the simulation. (Krizsan 2016.) In the figure 1 there is a simple example of a simulation created according to this description. The example simulation has 10 users, each sending 10 requests against an imaginary REST API.

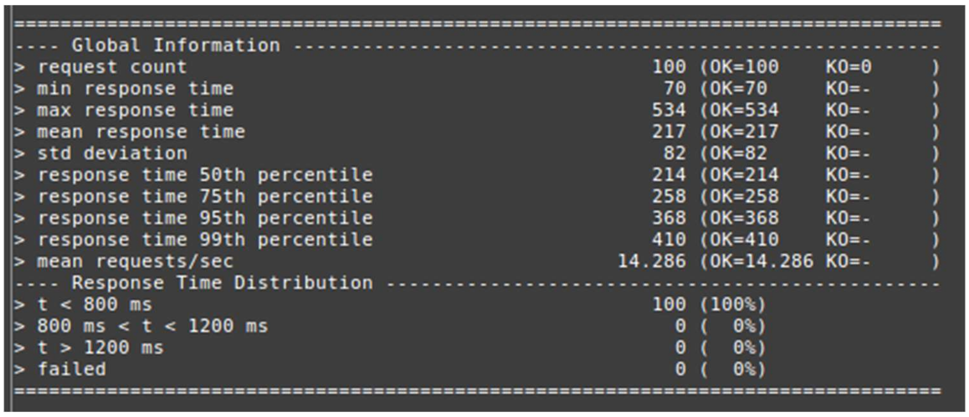


```scala
import io.gatling.core.Predef._
import io.gatling.http.Predef._

class example extends Simulation {
    var userCount = 10
    var rampUp = 10
    var requestsPerUser = 10

    private val httpProtocol = http
        .baseURL("http://localhost:8080")
        .acceptHeader("application/json")
        .headers(Map("user-id" -> "user",
            "sender-id" -> "tester",
            "locale" -> "en_US"))

    private val dataFeeder = csv("data.csv").circular

    private val scn = {
        scenario("Example test for imaginary rest endpoint")
            .feed(dataFeeder)
            .repeat(requestsPerUser) {
                exec(http("request_1")
                    .get("/rest/path/${id}")
                    .queryParam("startDate", "${startDate}")
                    .queryParam("endDate", "${endDate}"))
                    .feed(dataFeeder)
            }
    }

    setUp(scn.inject(
        rampUsers(userCount) over rampUp)
        .protocols(httpProtocol))
}
```

Figure 1. An example simulation for imaginary example class.

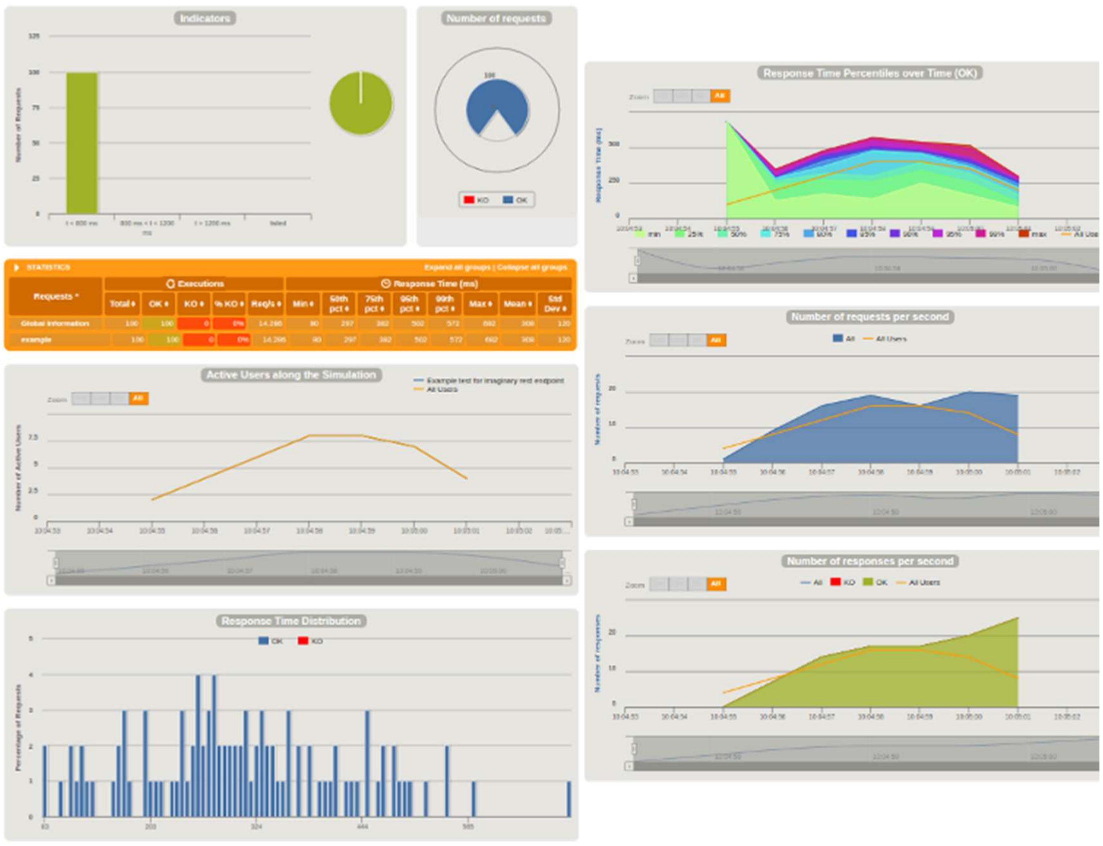
The benefits of using Gatling are its suitability for large scale testing, using external parameter data to populate the requests, validating the responses and the ability to generate informative result reports. (Tikhanski 2015.) External data sources can be read with feeders and by reading values with them we can pass unique parameters for the requests.

Gatling saves the results of a simulation into log-files and it creates HTML report pages for different requests. In the figure 2 there is a final summary of a simulation executed via command line, where a total of 100 requests were sent like in the previous simulation example. Figure 3 shows a HTML report generated by Gatling of this same execution.



```
================================================================================
---- Global Information --------------------------------------------------------
> request count                                        100 (OK=100    KO=0     )
> min response time                                     70 (OK=70     KO=-     )
> max response time                                    534 (OK=534    KO=-     )
> mean response time                                   217 (OK=217    KO=-     )
> std deviation                                         82 (OK=82     KO=-     )
> response time 50th percentile                        214 (OK=214    KO=-     )
> response time 75th percentile                        258 (OK=258    KO=-     )
> response time 95th percentile                        368 (OK=368    KO=-     )
> response time 99th percentile                        410 (OK=410    KO=-     )
> mean requests/sec                                 14.286 (OK=14.286 KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                           100 (100%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            0 (  0%)
> failed                                                 0 (  0%)
================================================================================
```

Figure 2. Example of Gatling command line report.



Indicators
Number of requests
KO
OK
Response Time Percentiles over Time (OK)
Number of requests per second
Active Users along the Simulation
Example test for imaginary rest endpoint
All Users
Number of responses per second
Response Time Distribution

STATISTICS — Expand all groups | Collapse all groups

Requests *	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global information	100	100	0	0%	14.286	80	297	382	502	572	692	308	120
example	100	100	0	0%	14.286	80	297	382	502	572	692	308	120

Figure 3. HTML report generated by Gatling

Gatling has support and extensions for Maven Archetype and Maven plugin, that allow you to integrate Gatling tests into a Maven project and execute them with mvn-commands. With Maven the project is easy to take and use with different environments and the whole project can be saved into version control. With Maven plugin the tests can also be easily executed in Jenkins amongst other tests ran against new builds. There is also a Gatling plugin for Jenkins, which searches log-files generated by the simulations from the Jenkins workspace. The plugin saves those files and generates graphs based on them. (Gatling [2] 2017.)

Basic knowledge in programming is beneficial when creating tests with Gatling. Scala as a programming language was completely foreign to me prior to this. After getting familiar with the basic syntax and Gatling, creating basic and some more complex tests became faster.

4.1.2 Apache JMeter

Apache JMeter is an open source Java application which is designed for testing functional behaviour and measure performance under load. It was originally created for web application testing, but it currently offers a wide variety of components and functions to use. (Apache [1] 2017.)

The JMeter package comes with .sh and .bat scripts, which are used to start up the graphical interface (GUI), which is used to create the actual tests. Because of this GUI test creation doesn't require programming skills and it also makes it easier to find possible flaws in the test logic. With GUI you can also add listeners to your test cases and get results of the test execution real-time, which makes creating and debugging the test cases easier. During the actual performance tests, the JMeter GUI should not be used in order to get the best performance as possible, as listeners and the graphical interface itself can consume a large part of the memory dedicated to JMeter JVM. (Apache [2] 2017).

JMeter saves the created tests into a JMX-file, which is basically a XML-styled file with the structure of the Test Plan, with the components and their configurations defined in it. Figure 4 shows a Test Plan created for the same imaginary example implementation as shown earlier with Gatling. In Jmeter all the tests are created and defined under the Test Plan-element. Test Plan works similarly as scenario in Gatling, it includes all components and logic for the test execution. Test Plan can hold multiple components and sub-components and their visibility may change depending on their relation to other components. In the example below only the basic and necessary components are added into the Test Plan.

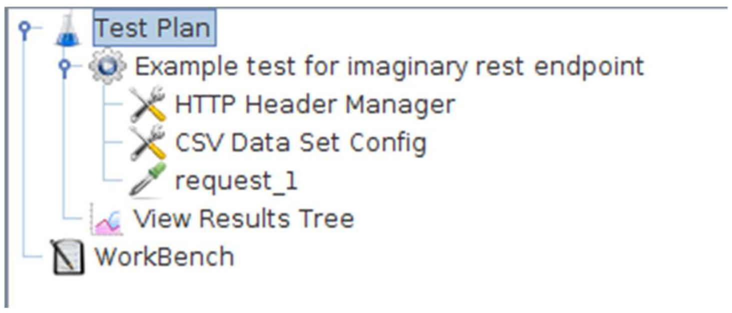


Figure 4. An example of Test Plan tree-structure

The Thread Group component defines the threads, which execute the steps that the group contains within it. In its configuration you can assign the number of users (number of threads), the rampup period (the time it takes in seconds for all the threads to start their execution), iterations per thread and optionally schedule the execution times or duration. (Apache [3] 2017.)

In the HTTP Header Manager component, you can define a list of headers that are sent with the HTTP requests. The requests themselves are configured within the HTTP Request component(s) (figure 4 request_1-component). You can set all needed properties for the tests within it, like server/IP, port, protocol, the method to be used, URL path, necessary parameters and for example the raw data needed for some POST-requests. (Apache [3] 2017.)

Generating random parameter data for the requests during performance test would take away lot of the processing capacity and memory we have available for our load tests. CSV Data Set Config-component allows us to read external csv-files one row at a time to parameterize our request. Each thread that starts sending a new request can be configured to look up certain variables from the csv-file. (Apache [3] 2017.) This allows us to create and send unique and parameterized request, without taking a major hit in performance.

In the picture 4 there is View Results Tree listener added into the test, which saves the requests that were sent and shows responses, processing time and other data of the request. (Apache [3] 2017) There are multiple other listeners as well that allow real-time monitoring of sending the requests from the UI. When running the tests from command line JMeter saves the basic information of requests into csv-result file. By default, it doesn't save all information about the requests, only information valuable in terms of measuring performance. It is possible to generate a separate HTML-report from this file with JMeter, similarly as with Gatling.

Like Gatling, JMeter can also easily be taken into use in any test automation environment with Maven. The tests can be assigned as a separate maven goal of some project or they can be kept in a project-folder themselves. There is also a Performance plugin for Jenkins that can be configured to create graphs and evaluate performance based on the csv-result file. It is possible to set threshold or absolute values that it monitors and can change the status of the job to success, unstable or failed accordingly. (Jenkins [1] 2017.)

4.1.3 Conclusion

Both tools can be used with or as a part of Maven project, in the comparison I created empty maven projects for both of the tools. The projects can be saved into version control as they are, even though the XML-format JMeter uses to save the test plans doesn't allow as substantial history tracking as Gatling's Scala.

Gatling's own Jenkins plugin doesn't allow us to compare the results from each test drive, it only produces reports and graphs of the test runs. With a performance tool called Taurus it is possible to create Gatling with its own script style, which then generates scala-files. Performance plugin supports the JUnit-results Taurus produces, which then can be used to create thresholds and compare results of test runs automatically. (Cohen 2017.) This requires installing Taurus into the test environment that is used, whereas with Maven all further installations are not required.

This was the biggest reason for me ending up using JMeter out of these two tools. After becoming familiar with the UI and the most important components I felt creating, testing the functionality and being able to monitor results in real time with the UI was much easier and faster than with Gatling. Apaches own documentation is very extensive and it is widely used tool and there are also a lot of external sources and information, blog posts and guides for solving problems you eventually might run into.

This doesn't mean that Gatling is a bad tool, it is still fairly new and will probably develop a lot over time. Creation of more complex and tests that exist of multiple components might be easier than doing the same with JMeters components. When testing REST APIs, the requests and configuration needed for them is quire straight forward, so in the end I valued JMeters UI that allows real-time testing and stock Performance plugin-support it has for Jenkins more.

4.2 REST API performance testing

The goal of the practical implementation in this thesis was to experiment with creating the tests and to be able to evaluate how they would suit for the company's test automation in practice. The setup for testing was experimented with REST architecture-based implementation that was developed for one customer project within Profit Software. No real performance or load tests had been carried out for this implementation prior to this thesis. It was important to be able to find issues that may occur when multiple users request the same API endpoint or resource at once, so these issues could be fixed prior to the release into production.

The customer project in question has an existing cloud-based test environment and within that one performance server with approximately half a million policies. In the future it should be possible to clone this performance server and deploy it in different test environments, where different tests including the REST performance tests could be executed against it. This environment has enough unique data for running the REST performance tests and it would be possible to deploy new cloned test environments where you could edit, create and delete data without making changes or destroying the original environment. The cloned environment would also provide a stable environment for testing, where the data and the parameters used for the tests would stay the same, making the results as comparable with each run as possible. The tests were to be created to suit running against this environment.

For the practical testing I created a small demo-environment for executing the performance tests. This setup had all the components needed to run and automate the tests installed and running locally on my computer. For this local testing the tests were carried out against a few selected APIs, that were seen as the most important and computationally heavy operations, such as requesting an account statement and calculating a new offer for example. The goal was to be able to find and pinpoint issues that may occur in functionality and to record and compare the performance of these selected APIs between results from different test runs.

In the following chapters I will go over all the steps in the process of creating, setting up and running the REST API tests in a local environment.

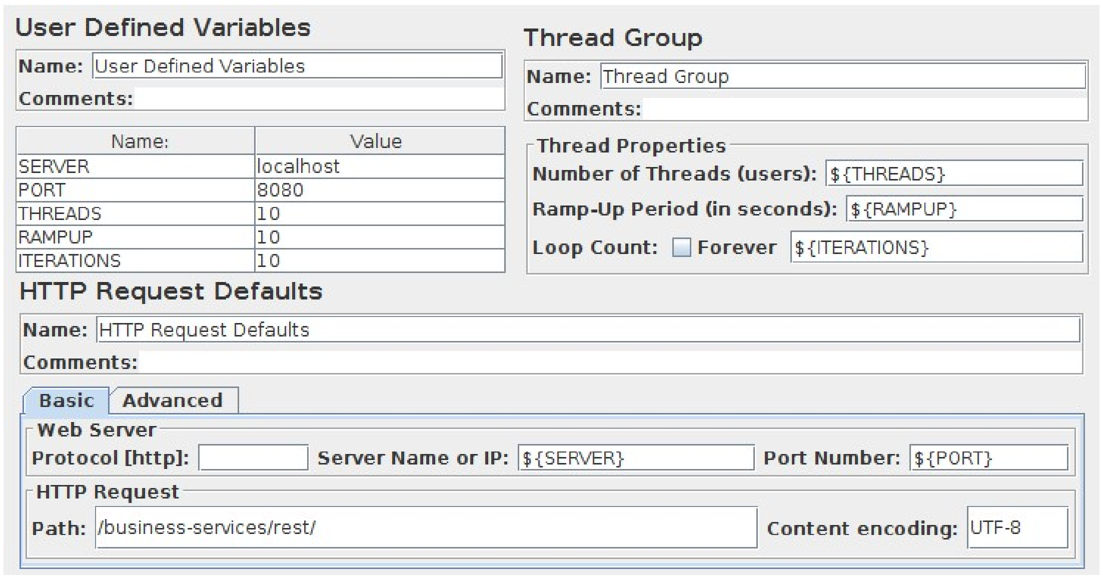
4.2.1 Creating the tests with JMeter

Some basics of creating test plans with JMeter and its components were already covered in the JMeter introduction chapter above. The example was very basic and only included absolutely necessary components for sending requests. In this chapter I'll go through the testing logic and components needed for creating and maintaining the REST API tests in JMeter.

When creating the tests for multiple REST APIs I decided to create separate Thread Groups for each API and selected the option to run them consecutively during the test execution. Each Thread Group will have one HTTP Request sampler in it, so for each iteration the thread will send one request. This means that during the execution each Thread Group and the threads defined for it will send requests against the same API endpoint. This allows us to generate more targeted and stable amount of artificial load against a single API with fewer amount of threads needed for the tests overall.

Having multiple Thread Groups makes it hard to quickly change the amount of threads or iterations per test, since you have to change that value individually for each group. The same problem comes if we need to change configuration of the HTTP samplers. For example, in order to change the amount of threads, iterations, server/IP or port we would have to edit these fields for each element separately. These inconveniences can be solved by using some basic JMeter components available.

JMeter's User Defined Variables component can be used to parameterize the Thread Group properties. It allows you to define an initial set of variables. These variables can be referenced throughout the Test Plan or inside the scope in which they're defined, which allows us to assign them as properties for the Thread Groups. Previously mentioned HTTP Header manager and HTTP Request Defaults components can also be added to the Test Plan level. With these you can define headers and default HTTP request properties that are common for all samplers in the Test Plan. (Apache [3] 2017.) For example, we can easily change the default request settings from local to test environment and change the amount of load for the tests with variables in the Thread Groups.


User Defined Variables

Name: User Defined Variables

Comments:

Name:	Value
SERVER	localhost
PORT	8080
THREADS	10
RAMPUP	10
ITERATIONS	10

Thread Group

Name: Thread Group

Comments:

Thread Properties

Number of Threads (users): ${THREADS}

Ramp-Up Period (in seconds): ${RAMPUP}

Loop Count: ☐ Forever ${ITERATIONS}

HTTP Request Defaults

Name: HTTP Request Defaults

Comments:

Web Server

Protocol [http]:

Server Name or IP: ${SERVER}

Port Number: ${PORT}

HTTP Request

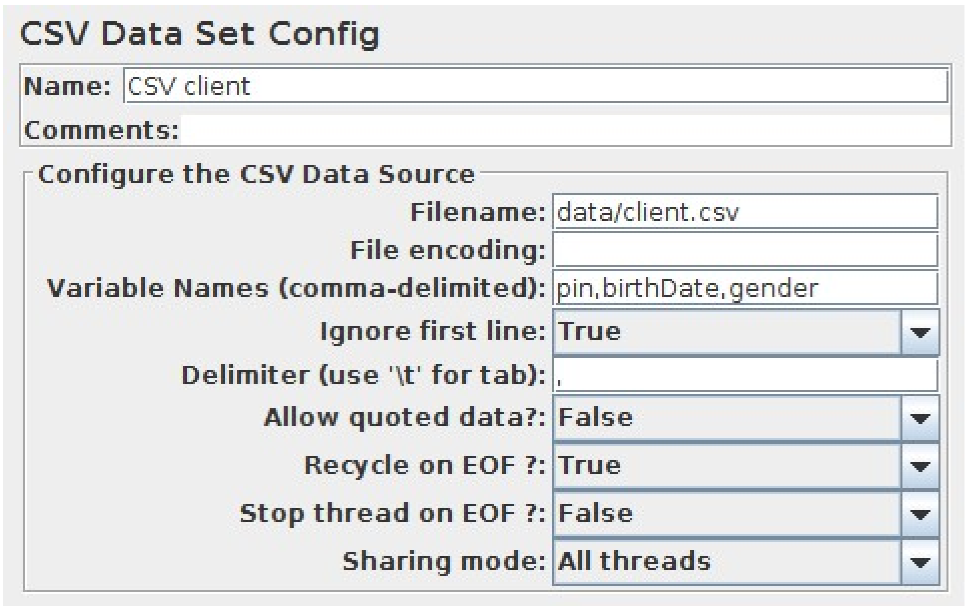
Path: /business-services/rest/

Content encoding: UTF-8

Figure 4. Example of using variables in Test Plan.

In our case we have multiple APIs that require some specific values to be passed with requests. Since generating these values at run-time is expensive in terms of CPU and memory, we want to use predefined datafiles to parameterize our requests. CSV Data

Set Config component can be used to read parameter data from files and load them into the memory and use them as variables at run-time. The data can be configured to be shared by all the threads so each thread that sends a request during the test execution will get the next row of variables from the CSV-files. There can be multiple CSV-files and their row sizes can vary since they can be configured to loop back to the beginning once the end of file is reached. Each file is configured in their separate config elements. The filename field uses path relative to the JMX-test file that uses it. (Apache [3] 2017.)


CSV Data Set Config

Name:	CSV client
Comments:	

Configure the CSV Data Source

Filename:	data/client.csv
File encoding:	
Variable Names (comma-delimited):	pin,birthDate,gender
Ignore first line:	True
Delimiter (use '\t' for tab):	,
Allow quoted data?:	False
Recycle on EOF ?:	True
Stop thread on EOF ?:	False
Sharing mode:	All threads

Figure 6. Configuration for client.csv datafile.

JMeter offers effective assertion tools for verifying the responses threads receive to their requests. While testing REST APIs these are not necessary since JMeter asserts the responses already based on the responses status code by default. During load test, adding assertions in our case only adds complexity and increases CPU and memory usage. When the response has a success status code we accept this and do not examine what the response holds within it any further. Client- and server-error status codes are marked as failures and we can investigate the cause for these further for example with listener components.

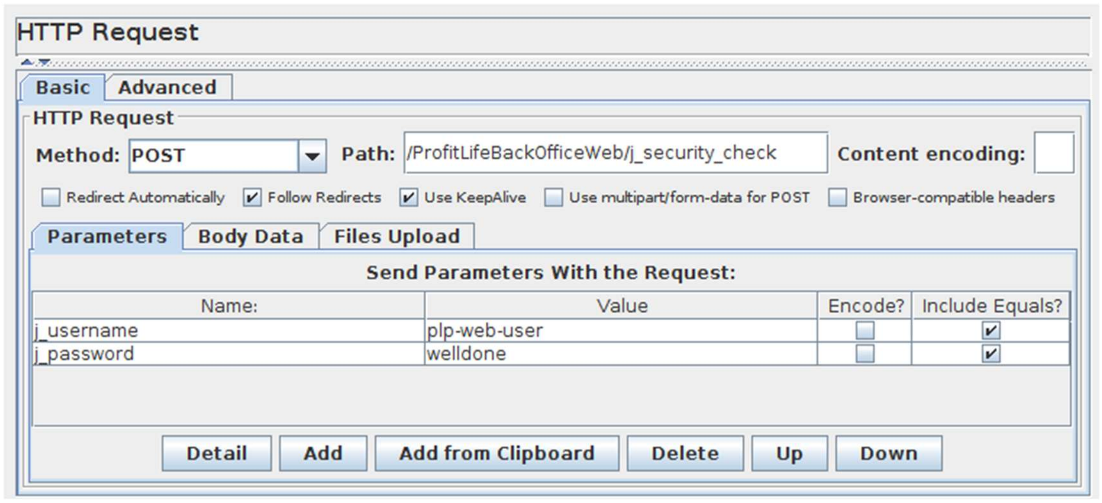
The listeners added to the Test Plan only work and log results individually when the test is run in the GUI. During non-GUI execution a single top-level CSV or XML listener is in use to log the most important data to a single result file. According to the best practices (Apache [2] 2017), we only want to use CSV format for our load tests in order to get the best performance out of JMeter. Unfortunately, the CSV log format doesn't save any request or response data. If there are failed requests during our test run, we could try and send these failed requests again and debug afterwards from the server logs what the cause for the failure was. For this, I added a Simple Data Writer for each Thread Group to write all request and response data to a file if an error occurs in that group. This allows us to get information about the failed requests that were sent, without suffering from a big performance hit. (Apache [4] 2017.)

4.2.2 Authenticating the REST calls

During creating these tests for the REST APIs there was a security level change committed into the customer projects PLP. In order to be able to send requests to the REST APIs the user now has to have the correct user rights and authentication. The authentication for JMeter testing can be achieved by first sending a login request to the PLP with a correct user and then using the cookie received in the response header to authenticate the rest of the requests.

In JMeter we'll use the HTTP Cookie Manager component for this. This component is designed to simulate a web browser, it automatically saves and sends cookies received by the thread that is sending the requests. The issue for artificial testing is that we generate new threads all the time and in different thread groups. The Cookie Managers cookies are individual and only available for the thread that received it to use. (Apache [3] 2017.) This means that every thread that is started would have to first send the login request before anything else, which would be really cumbersome to create and manage in JMeter. Luckily, we can use variables and properties to work around this issue.

First, a REST authentication Thread Group was added in as the first element to be executed during test run. You can use the setUp Thread Group component, which always gets executed first inside the Test Plan, but a normal Thread Group as a first group in the plan would work just as well. We configure it to generate one thread that is used to send the login request in order to receive the authentication as a part of the response. The authentication cookie received will be recorded by the HTTP Cookie Manager, which needs to be added inside this Thread Group. The test execution is made to mimic the login process via browser, sending a POST-request with the login credentials assigned as parameters to the elements of the login page, as shown in figure 7.


HTTP Request

Basic | Advanced

HTTP Request

Method: POST   Path: /ProfitLifeBackOfficeWeb/j_security_check   Content encoding:

[ ] Redirect Automatically [✓] Follow Redirects [✓] Use KeepAlive [ ] Use multipart/form-data for POST [ ] Browser-compatible headers

Parameters | Body Data | Files Upload

Send Parameters With the Request:

Name:	Value	Encode?	Include Equals?
j_username	plp-web-user	☐	✓
j_password	welldone	☐	✓

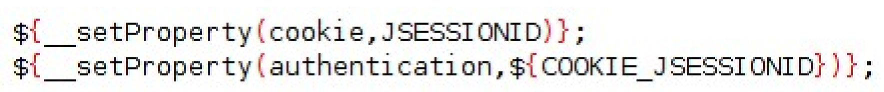
Figure 7. Login POST-request.

In order to use the authentication for all the requests, we need to change two CookieManager property configurations. These can be seen in the figure 8 of the POM below. CookieManager.save.cookies as true sets the response cookies as JMeter variables with a prefix "COOKIE_"- and CookieManager.check.cookies as false doesn't check the validity of the received cookies and saves every cookie it receives. (Apache [3] 2017.)

```
<configuration>
    <resultsFileFormat>csv</resultsFileFormat>
    <testResultsTimestamp>false</testResultsTimestamp>
    <propertiesUser>
        <server>${test.server}</server>
        <port>${test.port}</port>
        <threads>${test.threads}</threads>
        <rampup>${test.rampup}</rampup>
        <iterations>${test.iterations}</iterations>
        <duration>${test.duration}</duration>
        <CookieManager.save.cookies>true</CookieManager.save.cookies>
    </propertiesUser>
    <propertiesJmeter>
        <CookieManager.check.cookies>false</CookieManager.check.cookies>
    </propertiesJmeter>
</configuration>
```

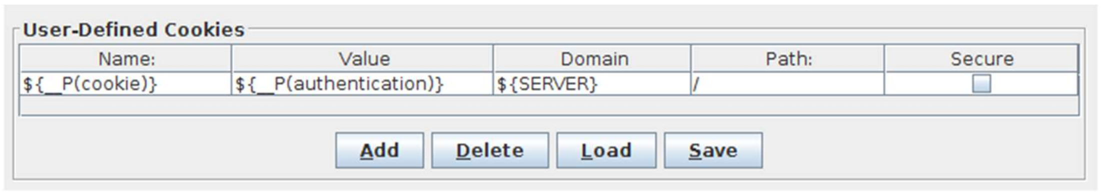
Figure 8. Final form of the POM-files configuration block.

The HTTP Cookie Manager saves the authentication cookie we get from the response as a variable, which are local to the thread. In order for other Thread Groups and threads to use it, we need to assign it to a JMeter property. A post-processor can be used to do so after the login request, the figure 9 shows a BeanShell post-processor script which is used to set these properties. The properties can then be used to configure User-Defined Cookies for every Thread Group in the Test Plan, as shown in the figure 10.



```
${__setProperty(cookie,JSESSIONID)};
${__setProperty(authentication,${COOKIE_JSESSIONID})};
```

Figure 9. Setting the cookies name and variable COOKIE_JSESSIONID as properties.

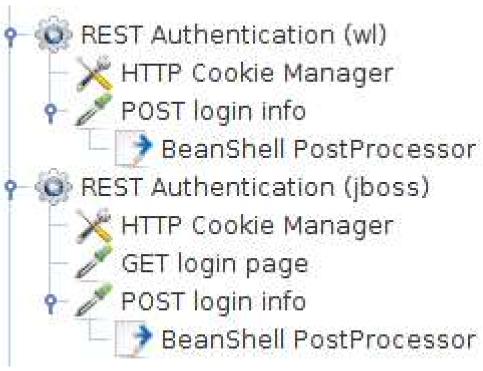


User-Defined Cookies

Name:	Value	Domain	Path:	Secure
${__P(cookie)}	${__P(authentication)}	${SERVER}	/	☐

Figure 10. Configuration of HTTP Cookie Manager to authenticate the requests of other Thread Groups and threads.

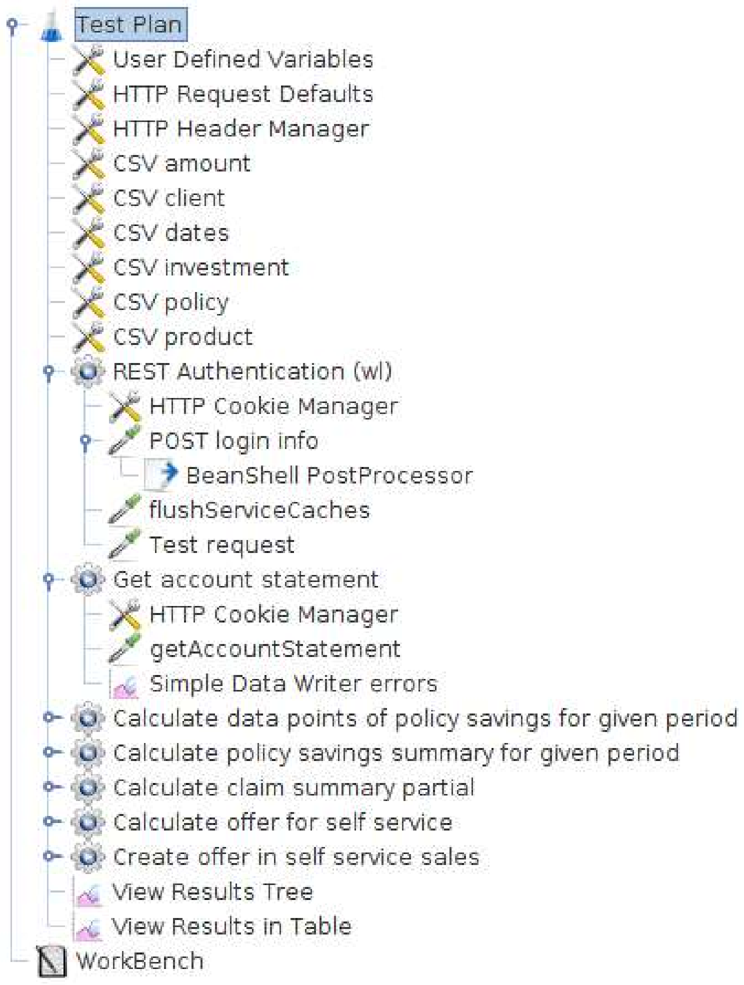
The customer project in question uses both WebLogic and JBoss/WildFly as application servers and the same configuration doesn't work for both. For JBoss the authentication cookie we need is called 'JSESSIONIDSSO', as opposed to WebLogic's 'JSESSIONID' shown above. By adding a GET-request to fetch the login page before sending our login POST-request and changing the values that we assigned with the post-processor to 'JSESSIONIDSSO' and '${COOKIE_JSESSIONIDSSO}', the same configuration works with JBoss as well.



```
REST Authentication (wl)
    HTTP Cookie Manager
    POST login info
        BeanShell PostProcessor
REST Authentication (jboss)
    HTTP Cookie Manager
    GET login page
    POST login info
        BeanShell PostProcessor
```

Figure 11. REST authentication configurations for WebLogic and JBoss.

In the current project structure there are two identical test files separated by suffix '-wl' and '-jboss', both having the correct configuration for the application server in question. Both tests work on their respected application servers, for maven execution the redundant one can be excluded from the test run in the POM.

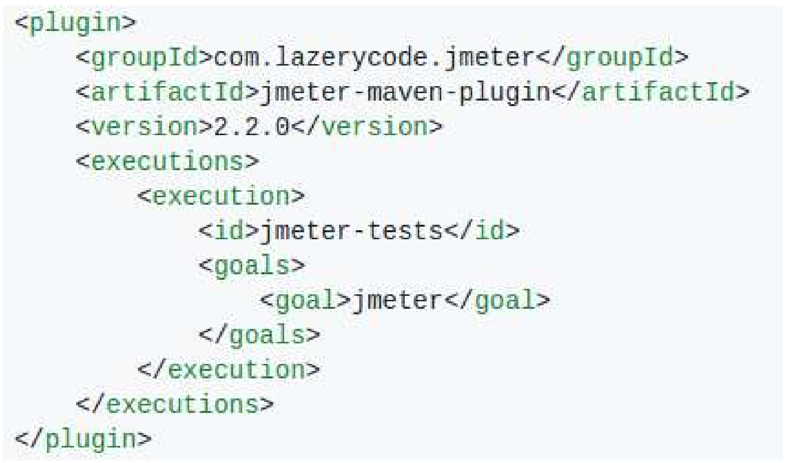


```
Test Plan
    User Defined Variables
    HTTP Request Defaults
    HTTP Header Manager
    CSV amount
    CSV client
    CSV dates
    CSV investment
    CSV policy
    CSV product
    REST Authentication (wl)
        HTTP Cookie Manager
        POST login info
            BeanShell PostProcessor
        flushServiceCaches
        Test request
    Get account statement
        HTTP Cookie Manager
        getAccountStatement
        Simple Data Writer errors
    Calculate data points of policy savings for given period
    Calculate policy savings summary for given period
    Calculate claim summary partial
    Calculate offer for self service
    Create offer in self service sales
    View Results Tree
    View Results in Table
WorkBench
```

Figure 12. Final Test Plan for WebLogic. Each Thread Group holds similar sampler as 'Get account statement' for corresponding API.

4.2.3 Maven plugin configuration

Maven plugin offers us a way to run the tests on any machine and any environment, without having to download and setup JMeter on the machine separately and by simply using maven commands to execute our tests. The maven version that was used for this thesis is 3.3.9 and the version for maven plugin is 2.2.0, which supports the newest version of JMeter.
Firstly, I created an empty project directory that followed mavens project structure and added the jmeter-maven-plugin into the build section of the POM-file. I also added a jmeter-folder to the structure and moved the created JMeter test files and data-folder there. During test execution the maven plugin fetches all JMX-files it can find under the src/test/jmeter-path and copies them into the target/jmeter/testFiles-directory. During the actual execution by default all of the JMX-files in the testFiles-directory will be executed. (Basic Configuration 2017.)



```
<plugin>
    <groupId>com.lazerycode.jmeter</groupId>
    <artifactId>jmeter-maven-plugin</artifactId>
    <version>2.2.0</version>
    <executions>
        <execution>
            <id>jmeter-tests</id>
            <goals>
                <goal>jmeter</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

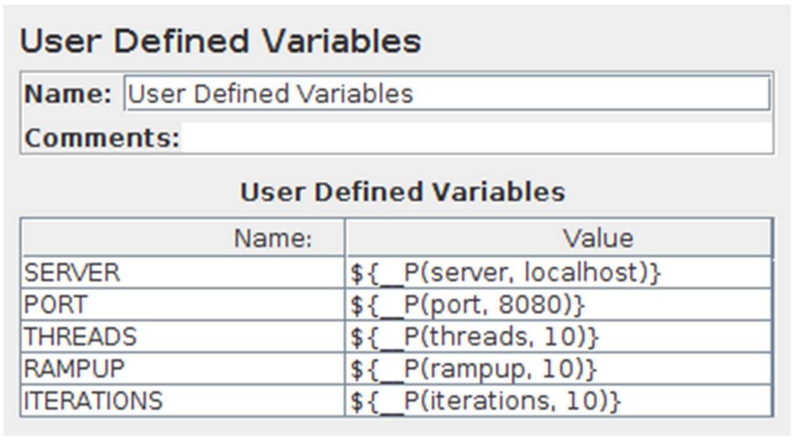
Figure 13. Maven-plugin configuration in the POM-file. (Basic Configuration 2017.)

You can also add some more specific configuration settings and variables for the test execution into the POM-file. It is possible to define certain tests that will be included or excluded from test run inside the configuration block. For performance testing and result logging the most important properties to set are resultsFileFormat and testResultsTimestamp. During the tests it is recommended to save the needed result data into a CSV-format, which is lighter than XML set by default. Also with the default configuration a timestamp gets added into the filename of the result file. This needs to be disabled in order for Jenkins Performance Plugin, which recognizes same tests by having the same name, to work correctly. (Advanced Configuration 2017; Apache [2] 2017.)

The User Defined Variables in the Test Plan can also be parameterized and overridden when executing the tests with maven. This makes it possible to change the variables for each test environment at run-time, without having to open and edit the variables in the test file itself. The maven properties can be predefined in the POM-file or they can be passed on as options when executing maven from command line. In the figures 14 and 15 below, all the variables that were hardcoded earlier in the JMeter GUI are now first set as maven properties. The plugin sets these values as JMeter user properties, which then are assigned in the JMeter GUI using the __P() function (short version of __property()) as variables. The function sets the property in question as JMeter variable, if the property is not found or you run the test using the GUI it defaults the variable to a set value. (Apache [5] 2017.)

```
<properties>
    <jmeter-maven-plugin.version>2.2.0</jmeter-maven-plugin.version>
    <test.server>127.0.0.1</test.server>
    <test.port>8080</test.port>
    <test.threads>10</test.threads>
    <test.rampup>10</test.rampup>
    <test.iterations>10</test.iterations>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>com.lazerycode.jmeter</groupId>
            <artifactId>jmeter-maven-plugin</artifactId>
            <version>${jmeter-maven-plugin.version}</version>
            <executions>
                <execution>
                    <id>jmeter-tests</id>
                    <phase>verify</phase>
                    <goals>
                        <goal>jmeter</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <resultsFileFormat>csv</resultsFileFormat>
                <testResultsTimestamp>false</testResultsTimestamp>
                <propertiesUser>
                    <server>${test.server}</server>
                    <port>${test.port}</port>
                    <threads>${test.threads}</threads>
                    <rampup>${test.rampup}</rampup>
                    <iterations>${test.iterations}</iterations>
                    <duration>${test.duration}</duration>
                </propertiesUser>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Figure 14. POM configuration that assigns the maven properties as JMeter user properties.



User Defined Variables

Name: User Defined Variables

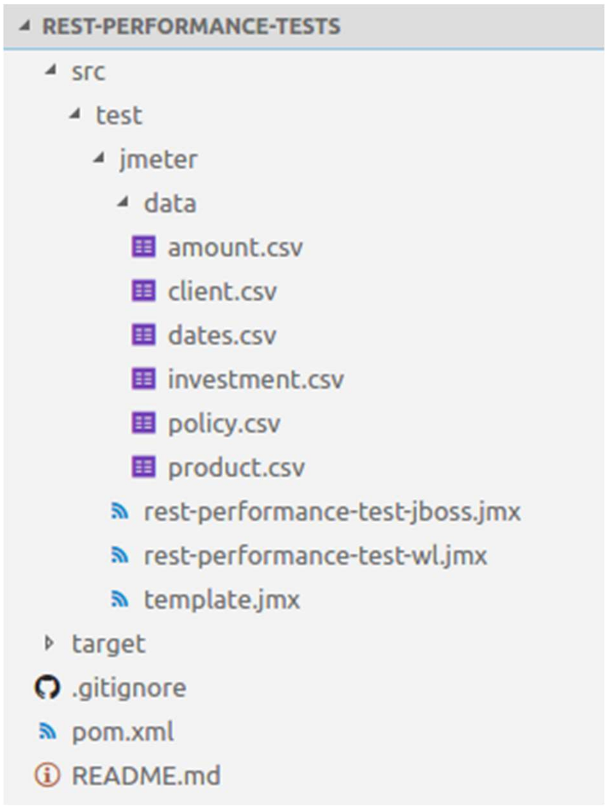
Comments:

User Defined Variables

Name:	Value
SERVER	${__P(server, localhost)}
PORT	${__P(port, 8080)}
THREADS	${__P(threads, 10)}
RAMPUP	${__P(rampup, 10)}
ITERATIONS	${__P(iterations, 10)}

Figure 15. JMeter properties assigned to local variables.

The figure 16 below shows us the final maven project structure for our tests. This whole project shown in the picture was also initialized and pushed into a git repository. The template.jmx has all the configuration files needed to execute the tests in it and it can be used to create and debug tests for new APIs. It is excluded from the test execution in the POM-file.



```
REST-PERFORMANCE-TESTS
  src
    test
      jmeter
        data
          amount.csv
          client.csv
          dates.csv
          investment.csv
          policy.csv
          product.csv
        rest-performance-test-jboss.jmx
        rest-performance-test-wl.jmx
        template.jmx
  target
  .gitignore
  pom.xml
  README.md
```

Figure 16. Complete maven project structure.

4.2.4 Jenkins configuration

For the purposes of executing the tests locally I installed the newest version of Jenkins directly to my computer, which at the time of writing this thesis was the version 2.60.1. I installed Jenkins directly from command line on my Linux machine as suggested in Jenkins documentation with 'sudo apt-get install jenkins'-command and reconfigured the default HTTP port from 8080 to 8081. (Jenkins [2] 2017.) The default plugins were installed as suggested by Jenkins. On top of those the Maven Integration plugin and Performance plugin that are needed when running JMeter tests with Jenkins were installed.

In my Jenkins I created a new maven project for the tests and configured it. I added the remote repository URL for the GIT project. In order for this to work I added the SSH key of the local jenkins user to have access into my personal projects in GitLab. By doing this Jenkins fetches the project and its contents from GIT into its workspace at the beginning of each test run.

In the build options I added pom.xml as root POM and the maven goal for this job as 'verify'. Verify is the maven lifecycle phase that the JMeter Maven plugin uses by default. You can also overwrite your POM-files maven properties as options on this line. For example, changing the amount of threads defined in the POM could be done by overwriting our test.threads property with "-Dtest.threads=100" option. Along with maven properties, JMeter settings can also be overridden if needed.

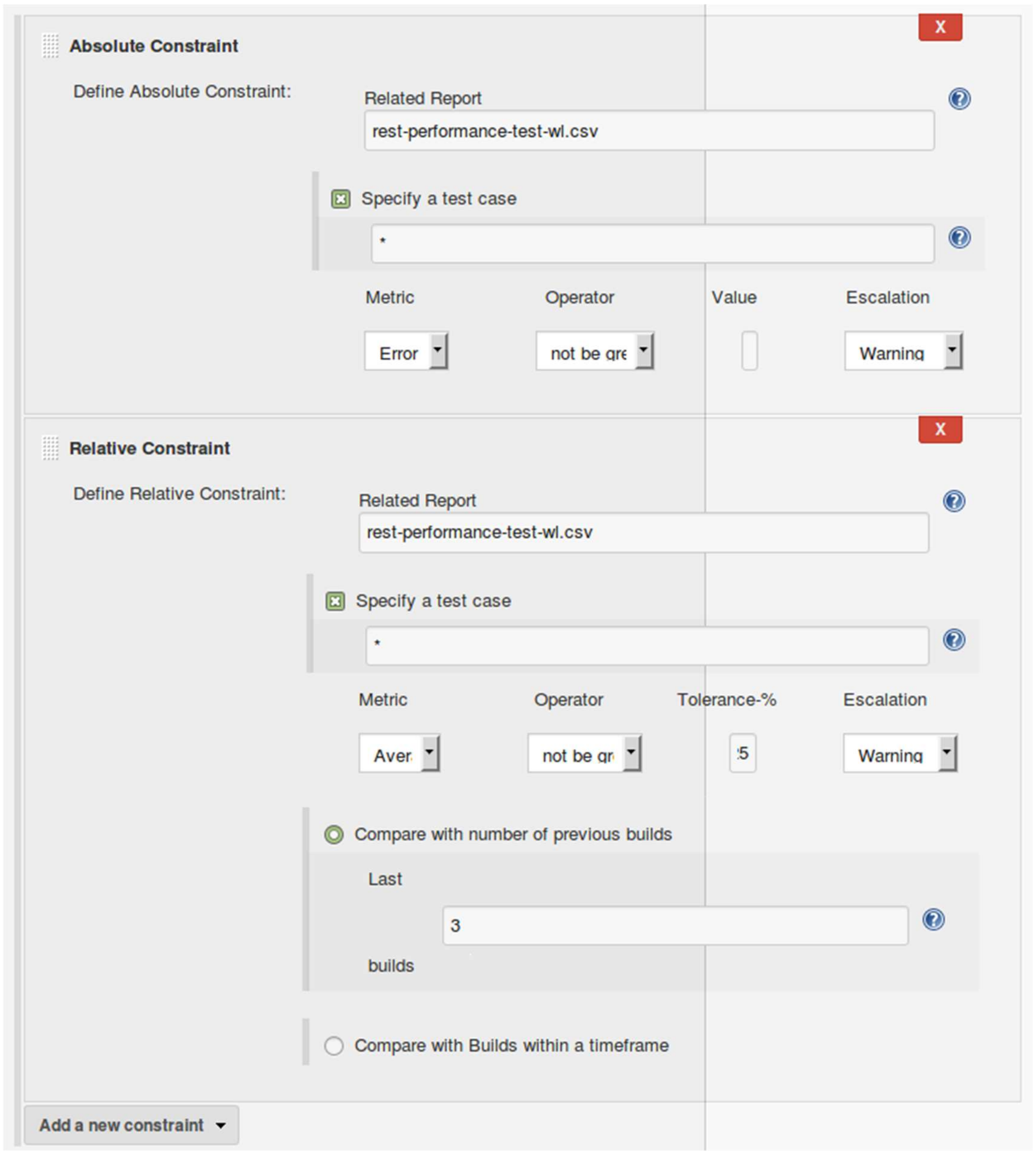
The performance plugin can be added to the tests from under the post-build actions and it can be configured to find all the JMeter result files from the target/jmeter/results-folder. It goes through the result files and reports sample sizes, response times and error percentages for each test case in each result file. Once you have ran your test at least twice, it automatically creates performance graphs for the test cases and under the build reports you can see performance comparisons against the previous test run. The performance plugin doesn't save the result files that it uses, only the calculated values for performance and data it needs for tracking. If the result files are needed they can be saved as artifacts.

4.2.5 Performance plugin configuration

The plugin offers more to the advantages for automation than just generated graphs and saving data of each test run. It can be used to measure and assert certain constraints, relative or absolute to compare the newly ran build against previous or set performance values of the test. With standard mode you can track and compare a single attribute and at the time of writing this there is a reported bug that doesn't allow reading more than one CSV-file for comparison. The expert mode allows you to have multiple and more specific constraints and that's the one we're using.

For my local environment I added few constraints to test these out. First one is an absolute constraint that checks whether there are any errors in the result files. We specify the filename for our test results and then select all the samplers with the '*'-wildcard option. We could also set individual constraints by specifying the name of a certain the test case, but applying same constraints for all cases is much faster to configure and easier to manage when there are multiple APIs tested within one test file. In figure 17 the absolute constraint has a value 0 in the Value field that doesn't scale properly, meaning that if there are any errors in the result file it will automatically change the status of the build to Unstable.

The relative constraint works in a similar manner, although it doesn't have a hardcoded value to compare to. It can be configured to compare the results of the executed test to the average of set number of previous builds or builds executed earlier during certain timeframe. This allows us to run the test multiple times at first and then compare the changes and fixes against the older test run averages. In the figure 17 the constraint is set to compare the average response time of all the test cases to their average in tree builds before and even if a single APIs average response time increase exceeds +25% it sets the build status to Unstable.


Absolute Constraint

Define Absolute Constraint:

Related Report: rest-performance-test-wl.csv

[✓] Specify a test case: *

Metric	Operator	Value	Escalation
Error	not be gre		Warning

Relative Constraint

Define Relative Constraint:

Related Report: rest-performance-test-wl.csv

[✓] Specify a test case: *

Metric	Operator	Tolerance-%	Escalation
Aver	not be gr	5	Warning

● Compare with number of previous builds

Last 3 builds

○ Compare with Builds within a timeframe

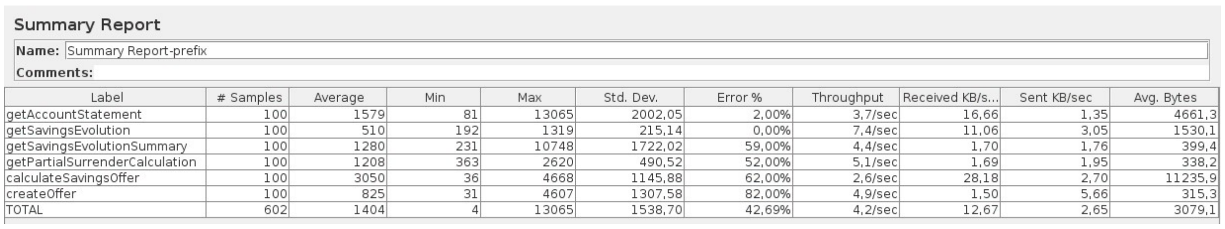
Figure 17. Jenkins Performance plugins expert mode configuration.

4.2.6 Testing the setup locally

With the setup created as explained above I was able to run small performance tests locally, similarly as they could be used as part of current test automation. I used some random exported savings and pension policies and imported them into my local dockerized PLP. I used the policies IDs, clients, existing investments etc. to populate the CSV-files with data so that each sent request would simulate real situation of client interacting with PLP through the REST APIs.

The load would be generated by 10 threads each sending 10 requests to the API that's being tested at a time, in total 100 requests would be sent to each API at a time. With this kind of local setup we we're able to find issues with parallel execution on some of the targeted interfaces. When the interfaces received multiple requests at the same time, the system was unable to handle and respond to all of the requests correctly. The validity of sent requests was ensured by using same exact csv data, but only using one thread to send the same 100 requests and the interfaces were able to handle and respond correctly.

The cause for these issues were tracked to be some non-thread safe functions used in the implementation. These caused issues when multiple requests were being handled at the same time. After being able find these issues the implementation was updated accordingly. Figures 18 and 19 show JMeters summary of these requests sent pre- and post-fixing.

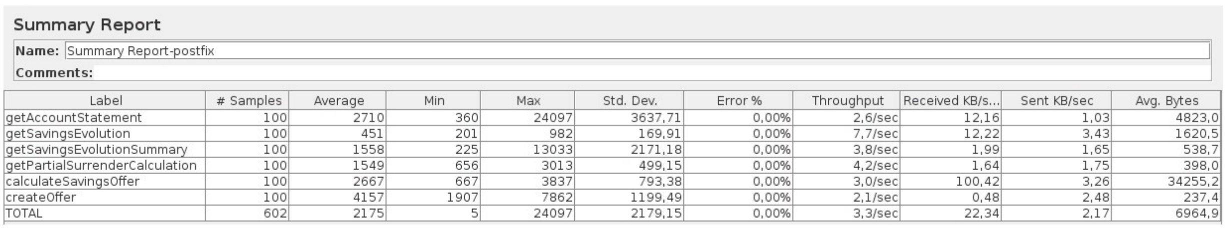

Summary Report

Name: Summary Report-prefix

Comments:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
getAccountStatement	100	1579	81	13065	2002,05	2,00%	3,7/sec	16,66	1,35	4661,3
getSavingsEvolution	100	510	192	1319	215,14	0,00%	7,4/sec	11,06	3,05	1530,1
getSavingsEvolutionSummary	100	1280	231	10748	1722,02	59,00%	4,4/sec	1,70	1,76	399,4
getPartialSurrenderCalculation	100	1208	363	2620	490,52	52,00%	5,1/sec	1,69	1,95	338,2
calculateSavingsOffer	100	3050	36	4668	1145,88	62,00%	2,6/sec	28,18	2,70	11235,9
createOffer	100	825	31	4607	1307,58	82,00%	4,9/sec	1,50	5,66	315,3
TOTAL	602	1404	4	13065	1538,70	42,69%	4,2/sec	12,67	2,65	3079,1

Figure 18. JMeters summary report prefix.


Summary Report

Name: Summary Report-postfix

Comments:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
getAccountStatement	100	2710	360	24097	3637,71	0,00%	2,6/sec	12,16	1,03	4823,0
getSavingsEvolution	100	451	201	982	169,91	0,00%	7,7/sec	12,22	3,43	1620,5
getSavingsEvolutionSummary	100	1558	225	13033	2171,18	0,00%	3,8/sec	1,99	1,65	538,7
getPartialSurrenderCalculation	100	1549	656	3013	499,15	0,00%	4,2/sec	1,64	1,75	398,0
calculateSavingsOffer	100	2667	667	3837	793,38	0,00%	3,0/sec	100,42	3,26	34255,2
createOffer	100	4157	1907	7862	1199,49	0,00%	2,1/sec	0,48	2,48	237,4
TOTAL	602	2175	5	24097	2179,15	0,00%	3,3/sec	22,34	2,17	6964,9

Figure 19. JMeter summary report postfix

We were also able to get some rough numbers on how much load generated by multiple users affect response times. Ultimately these numbers are highly affected by the environment in which PLP is running as well as the performance of the machine sending these requests. When running the tests in a specified environment, the results recorded should be comparable against other results recorded in the same environment with the same setup. Some margin for error should be taken into account depending on uncontrollable variables caused by possible issues in cloud environments and changes in latency between systems.

4.2.7 Moving the tests into test automation

The Test Plan created can be used to perform large scale performance and functionality testing against an environment that has a lot of policies in it. The policies can be used to populate the CSV-files so that we'll be able to send unique requests to the server, avoiding database and PLP's response caching that might occur with smaller data sets. At the time of writing this thesis, the performance server that the project has cannot yet be cloned for testing. Importing hundreds of policies for each individual test run would also be inconvenient and take too much time.

To solve this, I created a separate branch for the project for nightly build execution. The structure was kept the same, but in order to create simultaneous load I added a Synchronizing timer for each sampler in the Test Plan. All groups will have 10 threads, each sending one request to the server. The timer waits until the group has started up a set amount of threads and then releases them, causing all requests being sent at the same time as a spike. With this logic we were able to find the same issues as with larger data sets and samples as shown previously, but only having to import 10 savings and 10 pension policies into the test environment to create ten unique requests for each API.

Moving these tests into the existing test environment was fairly straight forward. Jenkins builds a new version of PLP every night and if it passes certain validation tests it will automatically be deployed on to the test servers. The Performance plugin and GIT configuration was the same as used locally. The only changes needed to be done were to use an existing script to clean the database and to include the policies and clients used for testing into the project and import them into PLP as a pre-step before executing the tests. Some minor changes and improvements were made to the testing logic as well, but we were able to run these tests and get results in our Jenkins.

Jenkins pre-step cleans the database and imports the needed 20 policies into the docker PLP for testing. This took under 3 minutes in all of my sample builds and the total time for the builds never surpassed 6 minutes. The time it takes to start up threads and send the requests will increase when there are more APIs added into the tests, but overall with synchronizing timer the tests can be executed in rather short time and should definitely be included in regular nightly build testing cycle.

5 CONCLUSION

The research and implementation done shows that testing the REST APIs could be taken as part of regular product testing now and in the future. Without testing, these kinds of issues that we have been able to find would have only presented themselves in production, where it would have been more challenging to debug the cause of these issues and they would have had an impact on the customers business.

The work on the customer project has been continued and currently almost all REST APIs implemented so far are included in the test siles. In the future as more APIs are implemented into PLP, creating performance tests for these will hopefully follow. JMeter and similar setup that's based on this thesis has also been taken to use in another project within the company to measure whether the performance of certain REST APIs match the requirements agreed with the customer.

There are still improvements that could be made and to consider with testing the REST APIs. The tests were carried out as black box testing and currently the tests created only use policies and send requests that fit a certain criterion but are otherwise randomly selected. Because of a small policy set used, there are different cases and policy types that might still cause new issues.

Currently the amount of load we can create with the tests is limited by the Jenkins machine running JMeter and how many threads it can handle. JMeter offers support for using multiple machines sending requests in master-slave configuration, but this has not been considered for testing as of yet. The distributed testing setup would be more challenging to run as an automated process. When at some point we need and want to use more load than our single machine can manage, it might be necessary to look into third party systems that provide cloud based distributed testing.

The performance numbers from test runs aren't fully comparable with each test run or to the results the customer will have in production. Currently the tests are using 10 threads. Although this can be easily increased, we can only estimate how many users will be using certain APIs in production, which may cause performance to decrease or new issues to appear when handling the requests. Test environments and Jenkins both are in cloud and there will always be some changes in the environment and latency that might affect and distort the results. The nightly build logic also sends only 10 requests per API, which might lead to situations where some issues might not appear because of such a small sample size.

Despite these issues mentioned, the results offer at least some baseline numbers to compare against our own similarly set up test environments and drastic changes in performance or success rate are still visible and can be acted upon. The setup can be further optimized and once we're able to run more comprehensive tests with a larger set of policies, some of the issues mentioned won't affect the results as much.

REFERENCES

Advanced Configuration. Updated 29.4.2017. Cited 26.7.2017. https://github.com/jmeter-maven-plugin/jmeter-maven-plugin/wiki/Advanced-Configuration

Apache (1). 1999-2017. Apache JMeter. Cited 25.7.2017. http://jmeter.apache.org/index.html

Apache (2). 1999-2017. Best Practices: 16.7 Reducing resource requirements. Cited 25.7.2017. http://jmeter.apache.org/usermanual/best-practices.html

Apache (3). 1999-2017. Component Reference. Cited 25.7.2017 http://jmeter.apache.org/usermanual/component_reference.html

Apache (4). 1999-2017. Listeners. Cited 5.8.2017. http://jmeter.apache.org/usermanual/listeners.html

Apache (5). 1999-2017. Functions. Cited 7.8.2017. http://jmeter.apache.org/usermanual/functions.html

Basic Configuration. Updated 6.5.2017. Cited 26.7.2017. https://github.com/jmeter-maven-plugin/jmeter-maven-plugin/wiki/Basic-Configuration

Cohen, Noga. 2.2.2017. 'How to Load Test with Gatling and Taurus'. Load testing. Cited 26.7.2017. https://www.blazemeter.com/blog/how-load-test-gatling-and-taurus

Fielding, Roy (1). 2000. 'Architectural Styles and
the Design of Network-based Software Architectures'. Cited: 27.10.2017. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Fielding, Roy (2). 22.11.2007. 'A little REST and Relaxation'. ApacheCon talk 2007. Cited 18.11.2017. http://roy.gbiv.com/talks/200711_REST_ApacheCon.pdf

Gatling (1). 2011-2017. Gatling documentation: Gatling. Cited 24.7.2017. http://gatling.io/docs/current

Gatling (2). 2011-2017. Gatling documentation: Extensions. Cited 25.7.2017. http://gatling.io/docs/current/extensions

Jenkins (1). Updated 5.5.2017. Performance Plugin. Cited 26.7.2017. https://wiki.jenkins.io/display/JENKINS/Performance+Plugin

Jenkins (2). 2017. 'Installing Jenkins'. Documentation. Cited 15.8.2017. https://jenkins.io/doc/book/getting-started/installing/

Krizsan, Ivan. 16.4.2016. 'Introduction to Load Testing with Gatling – Part 1'. Cited 24.7.2017. https://www.ivankrizsan.se/2016/04/16/introduction-to-load-testing-with-gatling-part-1/

Profit Software. 2014-2017. About Profit Software. Cited 23.10.2017. http://www.profitsoftware.com/why-profit-software/about-profit-software/

Selvaraj, Vinoth. 21.11.2016. 'Continuous Performance Testing – JMeter + Maven'. JMeter. Cited 5.8.2017. http://www.testautomationguru.com/jmeter-continuous-performance-testing-jmeter-maven/

Tikhanski, Dmitri. 29.9.2015. 'Open Source Load Testing Tools: Which One Should I Use?'. Performance testing. Cited 25.7.2017. https://www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use