ISSN: 0976-3104
SPECIAL ISSUE: COMPUTER SCIENCE

ARTICLE

GATLING AND VAADIN INTEGRATION IN CENTOS

Sukhendu Mukherjee*

Tiny Planet Inc. 5551 Orangethorpe Ave, Suite A, La Palma, CA - 90623, USA

ABSTRACT

This article describes how we can integrate Gatling with Vaadin. Gatling helps us to do load testing for any web application. We can use Gatling for Vaadin application for load testing. Gatling is a Scala-based load testing tool developed by the Gatling Corp. The tool itself is open source and can be found on GitHub. On top of the open part, an enterprise edition exists.

.

KEY WORDS
Gatling; Vaadin; Load Testing; Scala

INTRODUCTION

Load tests in Gatling [1-3] are written in Scala [4]. The API for writing those tests makes heavy use of the builder pattern and fluent interfaces. This might be a question of personal preferences but in my opinion this approach fits quite well. Especially, because no detailed Scala knowledge is necessary in order to write Gatling load tests. Therefore, Java developers should not be afraid of using Gatling.

A single load test in Gatling is called a scenario. Roughly, a scenario can be divided into three parts:

General configuration [1] (protocol, server address, encoding ...)
Steps to execute (open webpage, click this, enter that ...)
Scenario configuration (no. of total users, users over time ...)

The different parts will be explained in more detail in the following sections. But the possibilities for reusing different parts across tests should already be obvious.

Gatling currently provides support for HTTP protocols (including WebSocket and SSE) and JMS. Extending this functionality will be part of the next blog post. For the following example, we will rely on HTTP requests because they are the easiest to understand.

MATERIALS AND METHODS

Received: 4 April 2018
Accepted: 19 April 2018
Published: 22 April 2018

As a part of load testing we took our existing vaadin application [5] and followed below steps to integrate Gatling with Vaadin [5,6].

Install gatling
1. Download Gating bundle from Install Gatling url - https://gatling.io/download/
2. Just unzip the downloaded bundle to a folder of your choice.
3. Configure the proper encoding in the gatling. conf file

Start the recorder and configure it like in the screen shot

Use this $GATLING_HOME/bin/recorder.sh to start recorder.
. Once launched, the following GUI lets you configure how requests and responses will be recorded.
Set it up with the following options:

- In output folder we need to define the path where Gatling test cases will be generated.
- package name where scala file will be created under the defined package name.
- *Simulation file* name
- *Follow Redirects?* checked
- *Automatic Referers?* checked
- *Black list first* filter strategy selected
- *.*\.css*, *.*\.js* and *.*\.ico* in the black list filters

Configure the proxy in your browser

We need to configure [Fig-1] the proxy server to record the desired application activity in browser [2]. Please find below the proxy configuration screenshot. We need to make sure proxy setting port number and Gatling listening port should be same.

***Corresponding Author**
Email:
sukhendu.mukherjee@tinyplanetinc.com
Tel.: +1 214 862-6575

IIOAB JOURNAL
ISSN: 0976-3104

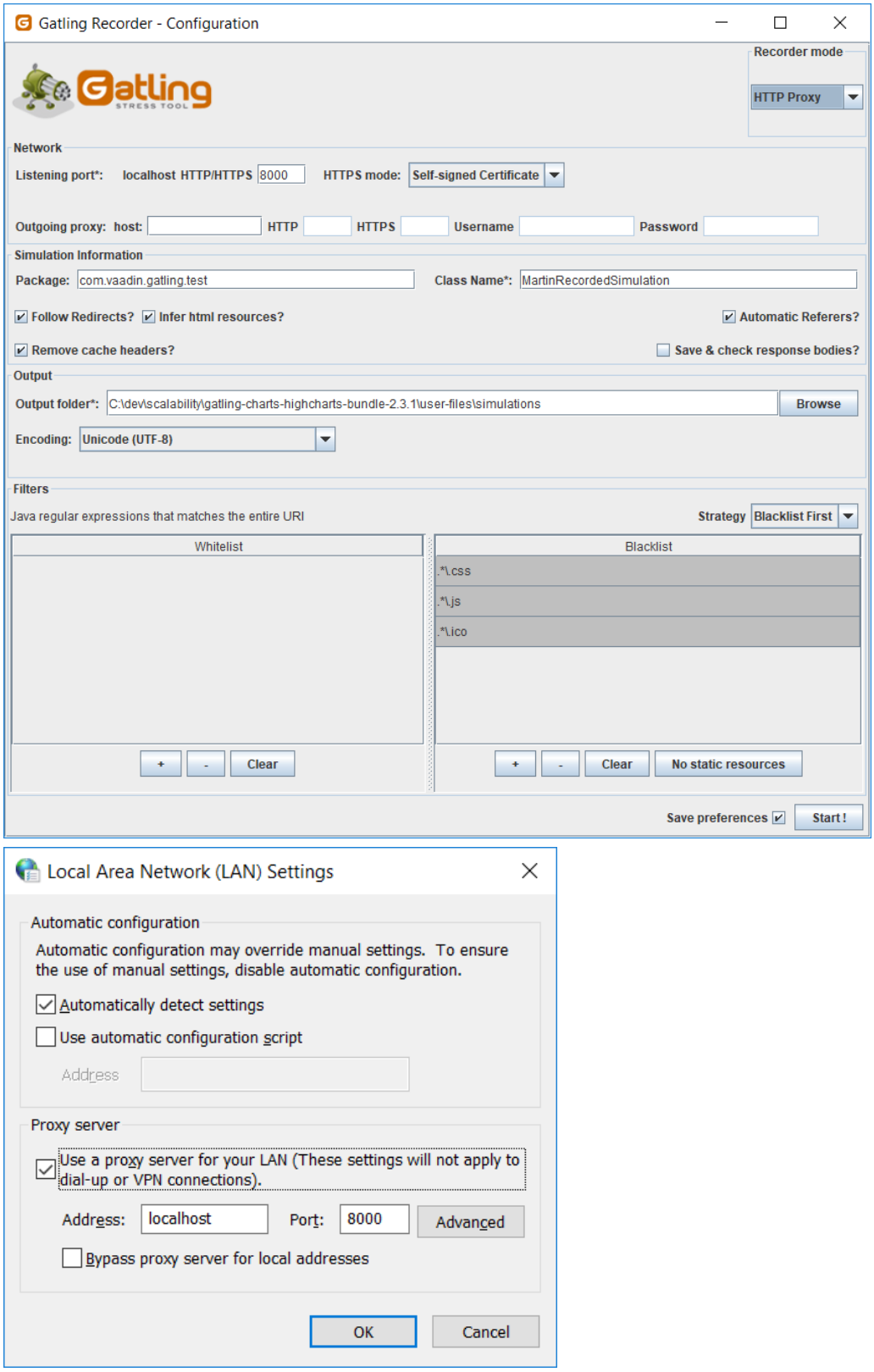

Fig. 1: Configure the proxy in browser

Start recording

Start the recording option from Gatling Recorder Configuration screen.

IIOAB JOURNAL
ISSN: 0976-3104

Start and Navigate the application

Now we need to navigate to the browser and do the activity in application to record the test cases we want to record.

Stop recording and Save

Once we done with the test cases and recording has been complete, we can stop the recording and save from Recorder Configuration screen and request*.txt file will be generated under request-bodies folder.

Copy the *.scala and *.txt files to the correct directories in application

Now we have completed recording and scala simulation file should be generated in the path we mentioned in Gatling Recorder Configuration [5] screen.We now need to copy the .scala and .txt files to the correct directories in application to have the test case ready.

Run the scalability test with the maven

Now we are good with run the scalability test with maven. We can use below command to run the scalability test cases from command prompt.
mvn -Pscalability gatling:execute Dgatling.simulationClass=com.vaadin.gatling.test.YourRecordedSimulation
We also need to configure pom.xml to get Gating working with maven. Below I have given a sample pom file mentioned Gatling configuration with vaadin [5, 6].

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.vaadin</groupId>
  <artifactId>gatling-vaadin-integration</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gatling-vaadin-integration</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <vaadin.version>7.3.2</vaadin.version>
    <vaadin.plugin.version>${vaadin.version}</vaadin.plugin.version>
  </properties>
  <repositories>
    <repository>
      <id>vaadin-addons</id>
      <url>http://maven.vaadin.com/vaadin-addons</url>
    </repository>
    <repository>
      <id>vaadin-snapshots</id>
      <url>http://oss.sonatype.org/content/repositories/vaadin-snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>vaadin-snapshots</id>
      <url>http://oss.sonatype.org/content/repositories/vaadin-snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
</pluginRepository>
  </pluginRepositories>
<dependencies>
 <dependency>
```

IIOAB JOURNAL
ISSN: 0976-3104

```
 <groupId>com.vaadin</groupId>
  <artifactId>vaadin-server</artifactId>
 <version>${vaadin.version}</version>
</dependency>
    <dependency>
 <groupId>com.vaadin</groupId>
  <artifactId>vaadin-client-compiled</artifactId>
  <version>${vaadin.version}</version>
</dependency>
 <dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-client</artifactId>
 <version>${vaadin.version}</version>
<scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.vaadin</groupId>
   <artifactId>vaadin-push</artifactId>
    <version>${vaadin.version}</version>
  </dependency>
   <dependency>
     <groupId>com.vaadin</groupId>
   <artifactId>vaadin-themes</artifactId>
  <version>${vaadin.version}</version>
 </dependency>
<dependency>
    <groupId>javax.servlet</groupId>
 <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
   <scope>provided</scope>
  </dependency>
 <dependency>
      <groupId>io.gatling.highcharts</groupId>
    <artifactId>gatling-charts-highcharts</artifactId>
   <version>2.0.1</version>
    <scope>test</scope>
</dependency>
</dependencies>
<build>

 <plugins>
  <plugin>
    <groupId>io.gatling</groupId>
   <artifactId>gatling-maven-plugin</artifactId>
  <version>2.0.0</version>
  <executions>
            <execution>
      <id>loadtest</id>
     <!--
        Configure the test to be run during integration-test
        phase automatically. Jetty server is configured to
        be running during integration tests in this example.
     -->
     <phase>integration-test</phase>
     <goals>
        <goal>execute</goal>
     </goals>
     <configuration>
        <!-- Default values -->
        <!--<configFolder>src/test/resources</configFolder-->
        <dataFolder>src/test/resources/data</dataFolder>
        <resultsFolder>target/gatling/results</resultsFolder>
        <requestBodiesFolder>src/test/resources/request-bodies</requestBodiesFolder>

        <simulationsFolder>src/test/scala</simulationsFolder>
     </configuration>
            </execution>
       </executions>
     </plugin>
     <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

IIOAB JOURNAL
ISSN: 0976-3104

```
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.7</source>
      <target>1.7</target>
    </configuration>
    <version>3.1</version>
  </plugin>
  <!-- As we are doing "inplace" GWT compilation, ensure the widgetset -->
  <!-- directory is cleaned properly -->
  <plugin>
    <artifactId>maven-clean-plugin</artifactId>
    <version>2.4.1</version>
    <configuration>
      <filesets>
        <fileset>
          <directory>src/main/webapp/VAADIN/widgetsets</directory>
        </fileset>
      </filesets>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.2</version>
    <configuration>
      <failOnMissingWebXml>false</failOnMissingWebXml>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.2.2.v20140723</version>
    <configuration>
                                        <scanIntervalSeconds>2</scanIntervalSeconds>
                                        <httpConnector>
                                                <port>${test.port}</port>
                                        </httpConnector>
                                </configuration>
    <executions>
      <!-- Configure jetty to start/stop the application
        for integration testing.
      -->
      <execution>
        <id>start-jetty</id>
        <phase>pre-integration-test</phase>
        <goals>
          <goal>run-exploded</goal>
        </goals>
        <configuration>
          <scanIntervalSeconds>0</scanIntervalSeconds>
          <daemon>true</daemon>
          <stopKey>STOP</stopKey>
          <stopPort>8866</stopPort>
        </configuration>
      </execution>
      <execution>
        <id>stop-jetty</id>
        <phase>post-integration-test</phase>
        <goals>
          <goal>stop</goal>
        </goals>
        <configuration>
          <stopPort>8866</stopPort>
          <stopKey>STOP</stopKey>
        </configuration>
                <execution>
              </executions>
            </plugin>
        </plugins>
      </build>
```

IIOAB JOURNAL
ISSN: 0976-3104

```
<profiles>
	<profile>
		<id>dev</id>
		<activation>
			<activeByDefault>true</activeByDefault>
		</activation>
		<properties>
			<test.port>8084</test.port>
			<jetty.stop.port>8090</jetty.stop.port>
			<test.hostname>localhost</test.hostname>
		</properties>
	</profile>
	<profile>
		<id>ci</id>
		<properties>
			<test.port>8082</test.port>
			<jetty.stop.port>8082</jetty.stop.port>
			<test.hostname>localhost</test.hostname>
		</properties>
	</profile>
</profiles>

</project>
```

Gatling has the following interesting features:

- Standalone HTTP Proxy Recorder, [5]
- Scala-based scripting,
- An expressive self-explanatory DSL for test development,
- asynchronous non-blocking engine for maximum performance,
- Excellent support of HTTP(S) protocols and can also be used for JDBC and JMS load testing,
- Validations and assertions,
- a Comprehensive HTML Report.
- Here is what a Gatling simulation looks like:

Gatling uses a more advanced engine based on Akka. Akka is a distributed framework based on the actor model. It allows fully asynchronous computing [5,6]. Actors are small entities communicating with other actors through messaging. It can simulate multiple virtual users with a single Thread. Gatling also makes use of Async HTTP Client.

The most simple HTTP test one can come up with is probably opening a web page and check that some content is being displayed. So, let's do that.
If it does not exist yet, please create a src/test/scala directory and use whichever package you prefer.

Every class has to extend io.gatling.core.scenario. Simulation in order to be recognized by Gatling. Additionally, the imports

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

are recommended. A Gatling module (here: core and HTTP) generally defines a class called Predef, which represents the central access point to that library. E.g. if we take a look at the io.gatling.http.Predef class, we can see that it just defines two types and extends io.gatling.http.HttpDsl, which provides the HTTP methods we need.

Validate the Scalability test results

Once maven command excuted successfully we can see the Gatling scalability test result in console and index.html file in /target/gatling/results/basicvaadinhellosimulation-1521470271678/index.html.This index.html file will *have the entire report of scalability testing.*

IIOAB JOURNAL
ISSN: 0976-3104

RESULTS

As a result we will be able to execute recorded scalability test cases and we can see the test report as shown if Fig-2. We can configure the number of users and will access the application in given timeframe in scala file.

```
// This uses more load, simulates 100 users who arrive with-in 10 seconds
        setUp(scn.inject(rampUsers(100) over (10 seconds))).protocols(httpProtocol)
```

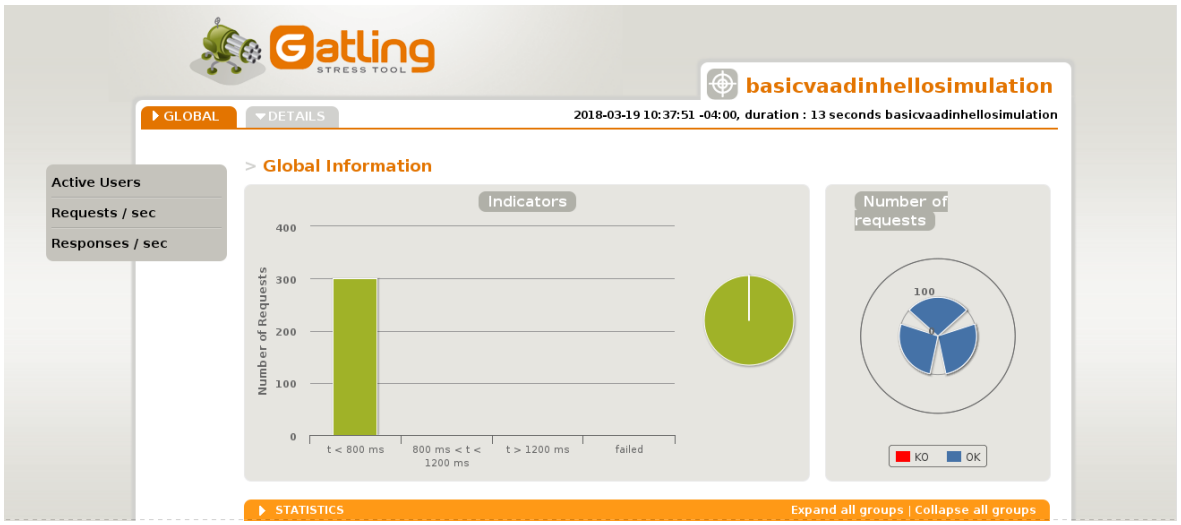

Fig. 2: Test report

CONCLUSION

Integrating Gatling for Vaadin [6] application to get load testing and scalability report for application to how the application will run in production and we can test with given number of users to see how its performing. Whichever way you chose to execute the tests, a results directory should have appeared. Within this directory another directory with the name of the scenario and a timestamp should be present. And lastly, within that one, an index.html file. This webpage contains all of the data that was collected by Gatling during the simulation, presented in a nice way.

CONFLICT OF INTEREST
None

ACKNOWLEDGEMENTS
None

FINANCIAL DISCLOSURE
None

REFERENCES

1. Gatling Corp. [2018] Gatling Documentation, Quickstart. Gatling Corp. Retrieved January 12, 2018.
2. Latinov L. [2017] Performance testing with Gatling. Automation Rhapsody. Retrieved September 1, 2017. "Scenario is a series of HTTP Requests with different action (POST/GET) and request parameters. Scenario is the actual user execution path. It is configured with load users count and ramp up pattern. This is done in the Simulation's "setUp" method. Several scenarios can form one simulation."
3. Rao SP et al. [2017). Gatling: A Lightweight Load Testing Tool. Performance Zone. DZone. Retrieved September 1, 2017. "Gatling consumes fewer system resources to run a load test than other options."
4. Latinov L. [2017] Performance testing with Gatling. Automation Rhapsody. Retrieved September 1, 2017. "Simulation" is the actual test. It is a Scala class that extends Gatling's io. gatling.core. scenario. Simulation class. Simulation has a HTTP Protocol object instantiated and configured with proper values as URL, request header parameters, authentication, caching, etc. Simulation has one or more "Scenario".
5. Latinov L. [2017] Performance testing with Gatling. Automation Rhapsody. Retrieved September 1, 2017. It is capable of creating immense amount of traffic from a single node.
6. *Vaadin Tutorial. https://vaadin.com/docs/v8/framework/tutorial.html*