Torturing Wildflies with Gatling Stress Tool

Table of Contents

Introduction

This document will describe how to perform **performance** and **load** tests using **Gatling** stress tool .

Objectives

- Introduce gatling tool
  - Setup Gatling in our CDI Crud application;
- Show the difference between performance and load testing
- Example on how to create the simulation;
- Execute performance tests after some system's improvements ;
- Torturing wildfly application server through load/stress testing.

A note on performance x load

Before reading this post it has to be clear the difference between both kind of tests:

- **Performance test**: This kind of test aims to constantly check/monitor the system's performance. The test must fail when the system performance (under normal peak) is not considered good or is below expectations (eg: response time or throughput got worse when compared to a predefined limit). Also note that this kind of the is **not** destructive and should be able to run more frequently than a load/stress test.

- **Load test**: is meant to test the system by constantly and steadily increasing the load on the system until it reaches a threshold limit. The goal of load testing is to expose defects (eg: memory leaks) and/or observe resources consumption when application is under abnormal situation like very high traffic.

Why Gatling?

The main advantage of **Gatling**, in my opinion, is that it is intuitive. When you write performance tests with the gatling API you're really describing user steps so it is easy to **understand**, **write** and **maintain** the test.

For example, when using JMeter you hardly will write a simulation without the help of its GUI to record the test. Also when a change is done in the simulation it is hard to know what was changed because its all XML.

> **TIP** XML is not a programming language.

Another advantage is that gatling is based on actors model, netty and NIO. So what that means?

- it is asynchronous - based on netty async client
- non blocking I/O - no need to wait for responses
- each user runs on its thread
- probably perform better then other stress tools
- produces more reliable simulation (it's more approximated to real application behaviour - mainly due to its threading model)

Also IDE support is a big advantage

Setup Gatling in our project

To configure the tool on our sample project we basically will use the gatling maven plugin as follows:

pom.xml

```
<plugin>
<groupId>io.gatling</groupId>
<artifactId>gatling-maven-plugin</artifactId>
<version>2.1.5</version>
<configuration>
<dataFolder>src/test/resources/data</dataFolder>
</configuration>
</plugin>
```

① External data (samples, user logins etc…) to be used in the simulation.

Also include the following maven dependency:

pom.xml

```
<dependency>
<groupId>io.gatling.highcharts</groupId>
<artifactId>gatling-charts-highcharts</artifactId>
<version>2.1.5</version>
<scope>test</scope>
</dependency>
```

Anatomy of a gatling test

- **Simulation** [*Simulation* is the name which is usually given to performance tests because they try to simulate the application's usage under real or even abnormal circumstances like e.g putting/simulating a lot of users using the app at the same time.]: Is the load/scalability/stress/performance test itself. It is made of multiple *scenarios*;
- **Scenario**: is the path a user takes to complete an action, a typical user behaviour. It is basically composed by http (it supports other protocols like websockets) requests;
- **Setup**: it is the configuration of our simulation. It will tell how much users or throughput each scenario will have. A setup is composed by a set of steps which will configure how users/throughput will be *injected*.
- **Assertions**: It is the expectations of our simulation, assertions will define if our test was successful or not. They can be declared at request level or globally.

Technically saying, a Gatling simulation is basically a scala file using an specific DSL which will perform calls (usually http) to application under test.

Let's see an example that shows a Gatling simulation.

A simple example

Here is a simple simulation which makes http call to our application REST API:

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

class CdiCrudSimulation extends Simulation{


  val httpProtocol = http <1>
    .baseURL("http://localhost:8080/cdi-crud/")
    .acceptHeader("application/json;charset=utf-8")
    .contentTypeHeader("application/json; charset=UTF-8")

  val listCarsRequest = http("list cars") <2>
    .get("rest/cars/")
    .check(status.is(200)) <3>

  val listCarsScenario = scenario("List cars") //<4> A scenario is a group of one or more
requests
    .exec(listCarsRequest)

  setUp( //<4> scenario setup
      listCarsScenario.inject(
        atOnceUsers(10),  <5>
        rampUsersPerSec(1) to (10) during(20 seconds),  <6>
        constantUsersPerSec(2) during (15 seconds))
       )
      .protocols(httpProtocol)  <7>
      .assertions( <8>
        global.successfulRequests.percent.greaterThan(95), // for all requests
        details("list cars").responseTime.mean.lessThan(50), // for specific group of
requests
        details("list cars").responseTime.max.lessThan(300)
      )

}
```

① Template for all http requests;

② Stores this request in a local variable;

③ Request assertion;

④ Scenarios configuration

⑤ Add 5 users at the same time (each on its on thread). They will fire one request (wait its response) each one.

⑥ scale from 1 to 10 users during 20 seconds (one user is added on each 2 seconds. On the last second the 10 users will fire requests simultaneously)

⑦ 2 users per second during 15 seconds (i fell quite dummy explaining this because the DSL is really **self explanatory**)

⑧ this section makes assertions on all or a group of requests

NOTE I've already talked about the REST API under test here

This simulation fires a total of 150 request in 34 seconds, here is the console output:

```
================================================================================
---- Global Information --------------------------------------------------------
> request count                                        150 (OK=150    KO=0     )
> min response time                                      8 (OK=8      KO=-     )
> max response time                                     38 (OK=38     KO=-     )
> mean response time                                    21 (OK=21     KO=-     )
> std deviation                                          5 (OK=5      KO=-     )
> response time 50th percentile                         22 (OK=22     KO=-     )
> response time 75th percentile                         24 (OK=24     KO=-     )
> mean requests/sec                                4.343 (OK=4.343  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                           150 (100%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            0 (  0%)
> failed                                                 0 (  0%)
================================================================================

Reports generated in 0s.
Please open the following file: /home/pestano/projects/cdi-
crud/target/gatling/results/cdicrudsimulation-1430707109729/index.html
Global: percentage of successful requests is greater than 95 : true
list cars: mean of response time is less than 50 : true
list cars: max of response time is less than 300 : true
```

And here are some graphical reports generated by Gatling:

Torturing Wildflies

Now the section that entitles this post, **Wildflies** is meant to be the plural of WildflyAS which will be the target of our simulation.

NOTE	The simulation was enhanced with more http calls to the rest API and also requests to the web application (which has only one page). It can be found here.
IMPORTANT	I will run the simulation and the application (the one deployed on Wildfly) on the same machine. This is **NOT ideal** cause both will compete for resources (CPU and memory) but is what I have at the moment and also simpler to show the concepts.

Software and harware

- Ubuntu 14.04/amd64;
- Java 8u40;
- Wildfly 9.0.0CR1;
- The application under test uses JavaEE7 stack, more details here;
- CPU i7-2670QM
- 8GB RAM

Performance tests

As described earlier, this kind of test must be able to be executed frequently so it can catch changes in our code that *possible* degradates the system performance.

It must not be destructive and IMO should execute on every signicative change, for example it could be part of a continous delevery pipeline.

NOTE	Jenkins has a Gatling plugin).

In following sub-sections we are going to make changes to our code and infrastructure (Wildfly) and run the simulation to see if the change was good or not.

First execution

Here is the result of a execution without changes to the code:

TODO link reports online (eg flickr)

Adding server cache to REST endpoints

Asynchronous REST Responses

Wildfly on mode domain

HTTP2 enabled on wildfly

Load tests

For load/scalability tests we will take another approach. We will perform the simulation over a longer period (eg:30 min) and will increasy users/requests slowly. At the same time we will attach a profiler and analyze resource comsuption like memory, garbage collection, CPU usage, threads etc...

References

1. http://pt.slideshare.net/swapnilvkotwal/gatling