Performance testing as a first-class citizen

Abhishek Tiwari

Citation: *A. Tiwari*, "Performance testing as a first-class citizen", Abhishek Tiwari, 2014. doi:10.59350/70r87-3td57

Published on: July 02, 2014

Today I want to talk about the performance testing. Automated web application testing is not very rare these days. This includes functional, unit and regression tests at minimum. A good web application delivery approach normally includes a continuous integration setup to run automated tests. Every time a new change is pushed to version control system, continuous integration server should run the automated tests and depending on test results deploy latest build in a staging or QA environment.

In last few year web has changed drastically. Until very recently web application performance testing was not considered as key requirement. Typical attitude to performance testing was reactive rather than proactive,. But now when performance is considered as one of the key user experience requirements there is a good reason to proactively test web application for performance and load. Performance testing can provide information about the behaviour of the web application during peak and off-peak conditions. According to various statistics, web application performance is affecting the bottom line of online businesses. Take a look on following interesting facts,

1. 40% of people abandon a website that takes more than 3 seconds to load. In 2013, during peak hours only 8% of top retailers managed an average download speed of three seconds or less on Cyber Monday.
2. The average web page has almost doubled in size since 2010. At same time mobile web browsing accounted for 30% of all web traffic in 2012 and is expected to grow to 50% by 2014.
3. During Cyber Monday due to sheer weight of traffic many ecommerce websites saw performance drop by more than 500% (usual 2.6 second load time to a 16 second load time when site traffic peaked).
4. When Mozilla reduced 2.2 seconds off their landing page load time, Firefox downloads increased by 15.4%. Whooping 60 million extra downloads per year. On average there was only ~1.5 seconds of page load time difference between downloaded and non-downloaders.

We can draw few conclusions here. First page load time is a key performance indicator. Second, it is not ok to serve a non-optimised desktop web page to a mobile device and do nothing about it. Third, stress or load testing is business critical.

Only by using a proactive and in-line automated performance testing approach businesses and delivery teams can avoid performance related issues.

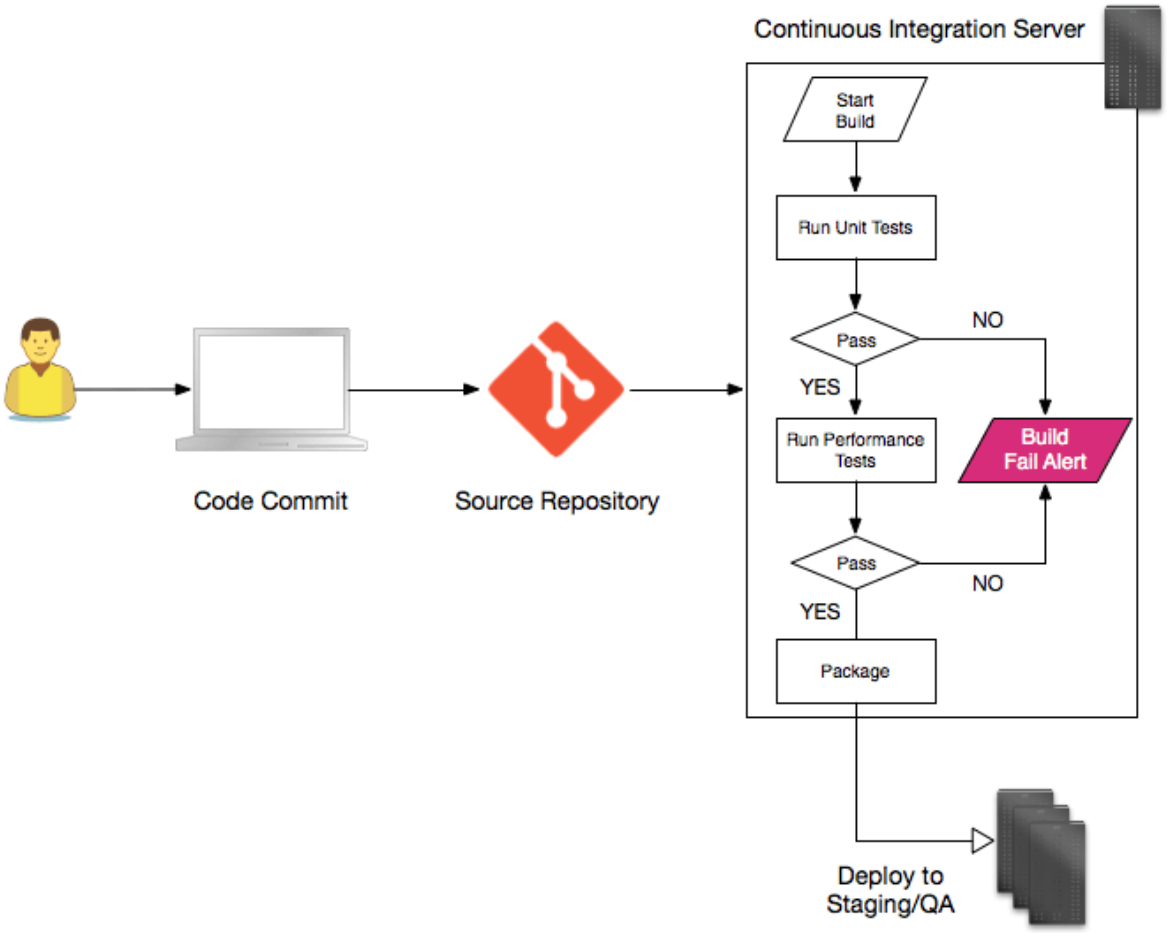

Figure 1: In-line performance testing workflow. Code commit to source repository triggers continuous integration which runs unit tests and performance tests

Load and Stress Testing

Open source tools with support for in-line automated performance testing is quite limited. Conventional performance testing tools or more precisely load testing tools such as Apache Benchmark(ab), perfmeter, Siege, http_load, JMeter, Grinder, boom, etc. are either hard to integrate with delivery pipeline or lack required features for in-line testing.

Locust a Python based open source load testing tool is my favourite when it comes to in-line automated performance testing. Using Locust you can define user behaviour with Python code. You can rush your web application with millions of simultaneous users in real-world like environment by running Locust distributed tests over multiple servers. Most importantly you can integrate these tests in your continuous integration pipeline. Locust measures request/second (average, minimum, max-

imum), total request, failed request, etc. You can see these reports in dashboard included as part of Locust.

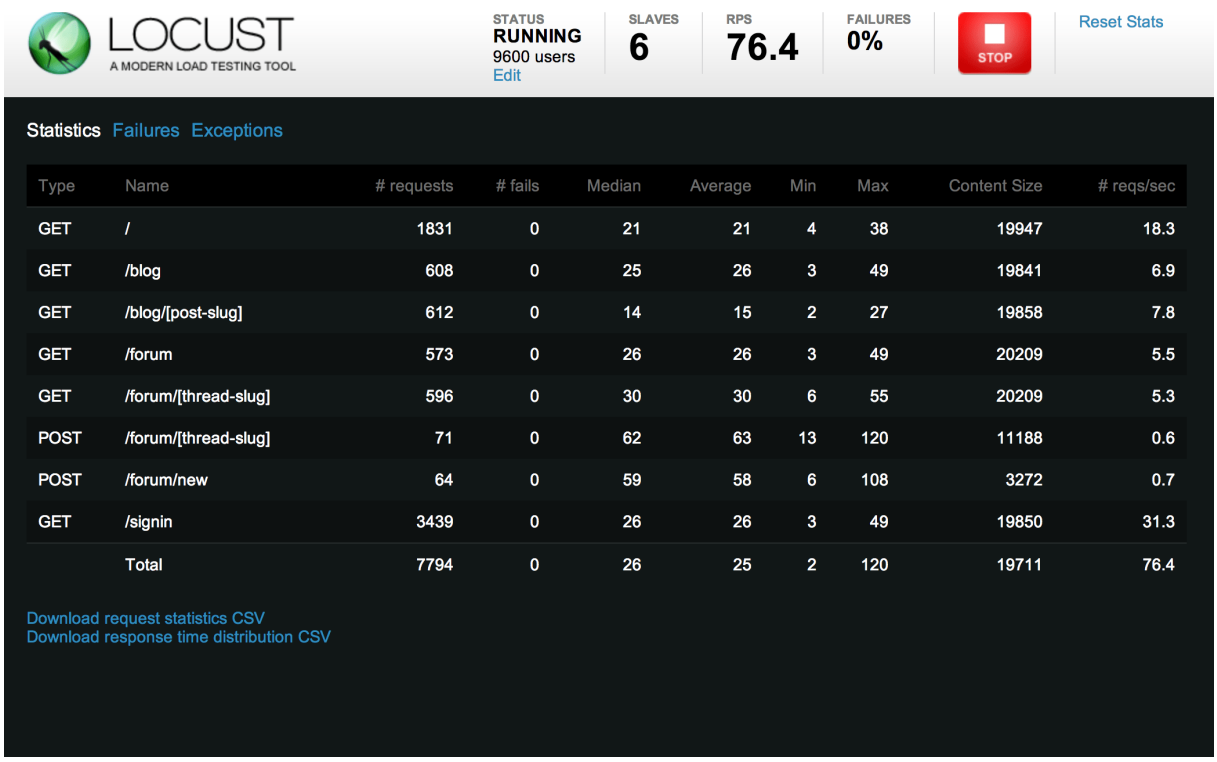

Figure 2: Performance Testing using Locust

Lets see one example of a Locust test file. A Locust test file normally defines a number of Locust tasks, which are normal Python functions with `@task` decorator. These tasks are gathered under a `TaskSet` class.

Simulated user is represented by `HttpLocust` class. This class defines how long a simulated user should wait between executing tasks, as well as `TaskSet` class that defines the execution behaviour of this test (i.e. simulated user's "behaviour"). `@task` decorator can also include optional `weight` which means each task will be weighted according to it's corresponding int value. To run the test execute `locust -f locustfile.py` from command line.

```
# locustfile.py
from locust import HttpLocust, TaskSet, task

class WebsiteTasks(TaskSet):
    def on_start(self):
        self.client.post("/login", {
            "username": "test_user",
            "password": ""
```

```
        })

    @task(weight=1)
    def index(self):
        self.client.get("/home/")

    @task(weight=2)
    def about(self):
        self.client.get("/about/")

class WebsiteUser(HttpLocust):
    task_set = WebsiteTasks
    min_wait = 5000
    max_wait = 15000
```

Page Speed Testing

Another interesting set of tools to measure and analyse the front-end performance or load time of web pages. My personal favourite in this category is WebPagetest which reports on page-level and request-level performance metrics such as Load Time, Fully Loaded, TTFB, Start Render, Speed Index, etc. Additionally, you can specify test location as well as browser.

WebPagetest provides both RESTful APIs as well as scripting and batch processing tools which you can use to integrate with your continuous delivery pipeline.

GTmetrix and SpeedCurve are alternative to WebPagetest. While SpeedCurve extends the WebPagetest, GTmetrix uses Google Page Speed and Yahoo! YSlow to grade your web page performance and provides actionable recommendations to fix these issues. GTmetrix also offers limited number free API calls per day.

Rise of SaaS performance testing tools

One of the key problem with Locust and other similar self-hosted tools is that you have to setup and manage clusters of machines for distributed performance testing. If your web application or website has a huge number of geographically diverse users, then a proper distributed performance testing requires sending client requests from multiple geographic regions (see my post on CDN performance and load testing). This may demand good upfront investment in infrastructure setup or appropriate automation to provision on-demand cloud infrastructure.

We can avoid all this trouble, by using a SaaS performance testing tool. There are many SaaS player in this space: Blitz.io , Tealeaf, SpeedCurve, LoadImpact, BlazeMeter, LoadStorm, etc. to just name a few. My favourite tool in this category is Blitz.io.

Blitz.io allows you do more than just load or performance testing. You can configure Blitz.io to do things like custom headers, authentication, cookie extraction, pattern, regions, schedule etc. Like other SaaS offering you can run multi-step and multi-region tests. All this, simply by using Blitz.io web interface.

Moreover, Blitz.io offers clients for Java, Maven, Node.js, Python, Perl, PHP and Ruby. With Blitz.io plugins for various continuous integration servers you can run load and performance tests as CI task. This means you can write performance tests in your favourite language or framework and integrate with continuous integration or build process. Here is one example of a simple Blitz.io test script,

```
from blitz.sprint import Sprint
from blitz.rush import Rush

def callback():
    print("SUCCESS")

def sprint():
    options = {'url': "http://example.com"}
    s = Sprint('youremail@example.com', 'aqbcdge-sjfkgurti-sjdhgft-skdiues
        ')
    s.execute(options, callback)

def rush():
    options = {'url': "http://example.com",
        'pattern': { 'intervals': [{'start':10, 'end':100, 'duration'
            :30}]}}
    r = Rush('youremail@example.com', 'aqbcdge-sjfkgurti-sjdhgft-skdiues')
    r.execute(options, callback)

sprint()

rush()
```

Here we are using Blitz.io Python client to run sprint and rush. Rush is Blitz.io way of generating load in predefined pattern. Spring is a single thread runner. You can think rush as collection of sprints.

Unit Testing Number of Queries

Now a days most of web applications are built on top of a MVC framework and applications generally use ORM for database queries and application level caching. Many unit testing or web frameworks now support `assert` against number of database queries. For example in Django one can asserts that a given function call with arguments means n number of database queries are executed .

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

If you are not sure about exact number of queries, you can still do fuzzy testing with assertNumQueries

by giving a range for number of queries. This is quite useful when queries are being cached. Following test will ensure that the number of queries is between 5 and 8.

```
class FuzzyInt(int):
    def __new__(cls, lowest, highest):
        obj = super(FuzzyInt, cls).__new__(cls, highest)
        obj.lowest = lowest
        obj.highest = highest
        return obj

    def __eq__(self, other):
        return other >= self.lowest and other <= self.highest

    def __repr__(self):
        return "[%d..%d]" % (self.lowest, self.highest)

class MyFuncTests(TestCase):
    def test_1(self):
        with self.assertNumQueries(FuzzyInt(5,8)):
            my_func(some_args)
```

In some cases it is also possible to assert if cache is being hit or if queries are being executed. For example, in NHibernate one can enable `generate_statistics` property in configuration,

```
<property name="generate_statistics">true</property>
```

and then use generated statistics for test query was cache hit or not.

```
// act
MappedEntity retrievedEntity = session.FindById(entity.Id);
long preCacheCount = sessionFactory.Statistics.SecondLevelCacheHitCount;
retrievedEntity = session.FindById(entity.Id);
long postCacheCount = sessionFactory.Statistics.SecondLevelCacheHitCount;
// assert
Assert.AreEqual(preCacheCount + 1, postCacheCount);
```

Closing thoughts

That's all for now. Like every other type of web application testing, the purpose of performance testing is to provide a high level of confidence to business and delivery team in a web application's ability - ability to respond quickly without compromising end-user's experience. More specifically application's ability to handle high load conditions, respond to various patterns of traffic and client side page load speed prior to going live. A proactive and in-line automated performance testing is way to go about it.