Continuous Performance

Loadtesten voor developers met Gatling

Het testen van de performance van een applicatie gebeurt vaak incidenteel: vóór elke grote release of na wijzigingen, die vermoedelijk impact hebben op de performance. Zo'n aanpak heeft een aantal nadelen: je komt vaak pas laat achter performance-issues en het is lang niet altijd bij voorbaat duidelijk of een wijziging significante impact op de performance zal hebben. Zou het niet veel prettiger werken als je continue feedback krijgt op de performance van je applicatie? In dit artikel lees je hoe je dat voor elkaar krijgt.

We gaan eerst nog even terug naar de problemen met incidenteel performance-testen. Een periodieke test (bij elke grote release) is nog wel te verdedigen. Het is vervelend dat je pas aan het eind van een development-traject achter performance-issues komt, maar je komt er in ieder geval achter. Een aanpak die per definitie problemen oplevert, is ad hoc inschatten of voor een bepaalde wijziging een performancetest nodig is. Die inschatting is namelijk redelijkerwijs niet altijd goed te maken. Een aanpassing in een algoritme van een methode, die op meerdere plaatsen gebruikt wordt, kan ervoor zorgen dat heel andere functionaliteit performancegevolgen ondervindt dan je zou verwachten. Een ander voorbeeld is een webapplicatie met een rijke frontend, waarin een aanpassing wordt gedaan, die ervoor zorgt dat een REST endpoint op de backend bij gelijkblijvend gebruik ineens veel vaker wordt aangeroepen, waardoor de load per gebruiker op de server toeneemt. Er is niets aan de backend code gewijzigd, maar de performance keldert.

Incidentele tests zijn meestal ook tests die iedere keer weer compleet opnieuw opgezet moeten worden. Dat is vaak veel werk: het bedenken van scenario's, opnemen van de interactie tussen browser en applicatie en handmatige aanpassingen aan het opgenomen script. Wanneer de applicatie niet aan de gestelde performance-eisen voldoet, volgt een zoektocht om de aanpassing te vinden, die een belemmering vormt voor de performance van de applicatie. En die kan soms weken of zelfs maanden geleden gedaan zijn.

Performancetesten bij Continuous Delivery

Bovenstaande noeste handenarbeid was jaren geleden misschien acceptabel bij een watervalproces, waarbij er een paar keer per jaar een nieuwe applicatieversie werd opgeleverd. Maar met de huidige kort-cyclische Scrum-processen in combinatie met Continuous Delivery en DevOps - die ervoor zorgen dat een nieuwe applicatieversie eens in de paar weken, paar dagen of zelfs een aantal keer per dag naar productie wordt gebracht - is het niet meer met de hand bij te houden.

Om echt verder te komen, moet je het testen van de performance van je applicatie inbedden in je dagelijkse ontwikkelproces, op hetzelfde niveau als het uitvoeren van unittests en integratietests en met hetzelfde niveau van automatisering. Net als bij je andere tests moeten developers bij iedere applicatiewijziging zo snel mogelijk terugkoppeling krijgen over de impact op de performance van de applicatie. Belangrijk aandachtspunt daarbij is dat je ook moet nadenken over het geautomatiseerd (her) opnemen van performancetests. Als je dat niet doet, dan heb je ofwel tests die niet up-to-date zijn met meest recente applicatieversie, ofwel je hebt alsnog een hoop handwerk aan het opnemen van de testscripts.

Om continue feedback op performance te verkrijgen, experimenteren we bij een klant momenteel met een opzet om performance-tests mee te laten draaien op testomgevingen. Hierbij kunnen we over de tests heen een goed beeld krijgen van het verloop van de performance van de applicatie in een bepaalde periode. Dit doen we geautomatiseerd met behulp van Gatling.

Bert Jan Schrijver is Software craftsman bij JPoint. Hij is erg geïnteresseerd in Java, Continuous Delivery en DevOps.

Tim van Eijndhoven is Software craftsman bij JPoint. Hij houdt zich vooral bezig met Java, applicatie-architectuur, software security en open source.

Gatling

Gatling is een open source load testing framework, gebouwd met Scala, Akka en Netty.
Het ondersteunt de meeste gangbare features van moderne browsers, zoals caching, cookies, redirects en websockets. Gatling is zelf echter geen browser: het draait geen JavaScript, past geen CSS toe, reageert niet op UI events, maar het opereert op HTTP-protocol-niveau.

Gatling biedt een DSL (Domain Specific Language) voor het definiëren van testscripts en gebruikt Akka en Netty om asynchrone non-blocking HTTP-operaties te doen. De Gatling distributie bestaat uit twee tools: Gatling zelf, die de testscripts uitvoert en de Gatling recorder, die gebruikt kan worden om een testscript in de Gatling DSL te genereren. De recorder kan als HTTP-proxy fungeren om zo alle requests en responses van een browser om te zetten naar een testscript, of door een vooraf opgenomen set aan HTTP requests en response in HAR-formaat (HTTP archive) te converteren. Een HAR is feitelijk een JSON-representatie van een set requests en responses. Gatling zelf kan gestart worden als losstaand programma of vanuit buildtools als Maven en Gradle.

De Gatling DSL

De Gatling DSL is Scala-code, aangevuld met een verzameling hulpmiddelen voor het vastleggen van performancetests. In **Listing 1** zie je een eenvoudig Gatling testscript dat requests op Google afvuurt. Deze test vraagt de Google startpagina op en voert daarna een zoekactie uit. De responses worden gecontroleerd op statuscode en op inhoud. Het script wordt gestart door 10 gelijktijdige gebruikers, verdeeld over een periode van 5 seconden.

```
class MyFirstSimulation extends Simulation {

 val httpProtocol = http
   .baseURL("https://www.google.nl")
   .userAgentHeader("Gatling!")

 val myFirstScenario = scenario("My first scenario")
   .exec(http("Open the Google start page.")
   .get("/")
   .check(status.is(200))
   .check(css("input[name='btnG']", "value").is("Google zoeken")))
   .pause(1)
   .exec(http("Perform Google search on 'gatling'.")
   .get("/search?q=gatling")
   .check(status.is(200))
   .check(regex("<title>gatling - Google zoeken</title>")))

 setUp(myFirstScenario
   .inject(rampUsers(10) over(5 seconds)))
   .protocols(httpProtocol)

}
```

Listing 1

De Gatling DSL kent een aantal concepten. Een *scenario* is een testscript dat bestaat uit een aantal teststappen: HTTP-requests. Op elke HTTP-response kun je *checks* doen om bijvoorbeeld de HTTP-statuscode of de inhoud van de response te controleren. Via checks kun je ook data vanuit een response opslaan om bij een volgende request weer te gebruiken. Je kunt *feeders* gebruiken om data (bijvoorbeeld gebruikersnamen) vanuit een bestand, database of vanaf een URL in je script op te nemen.

Wanneer je een scenario start *inject* je een vastgesteld aantal gebruikers via een bepaald patroon. Hierboven is dat een *rampup*: over een periode van 5 seconden wordt het script door 10 gebruikers gestart. De Gatling DSL bevat daarnaast ondersteuning voor testen via websockets en JMS queues. Voor een volledig overzicht, zie [3] bij Referenties.

HAR

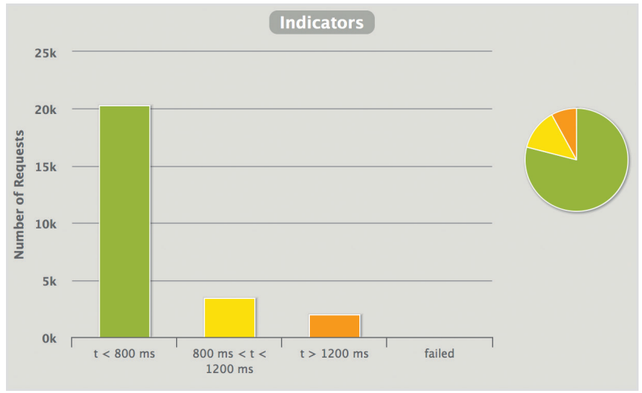
HTTP Archive (HAR) is een JSON-gebaseerd formaat om gegevens over interacties tussen de browser en de HTTP-server vast te leggen. Developer-tools in Chrome en Firebug bieden de mogelijkheid om het opgenomen netwerk-verkeer in HAR formaat te exporteren.

Opnemen van scripts

We hebben ervoor gekozen om onze testscripts niet zelf te schrijven, maar om ze op te nemen vanuit een browser. Hierdoor hebben we een zo representatief mogelijk beeld van de requestflow tussen onze frontend (AngularJS) en backend (Vert.x). Dat opnemen doen we momenteel met de hand, maar we zijn van plan om dit te automatiseren op dezelfde manier als onze automatische functionele tests, namelijk met FitNesse en BrowserStack.

Voor het opnemen van scripts gebruiken we bewust niet de proxy-functionaliteit van Gatling, maar HAR als tussenformaat. Dat zorgt ervoor dat je alleen een browser nodig hebt om een testscript op te nemen. Het opnemen van een script en het omzetten naar een Gatling DSL gebeurt in 2 verschillende stappen en kan dus door 2 verschillende personen (bijvoorbeeld een tester en een developer) worden uitgevoerd.

Het opnemen van een script is dan niet meer dan het doorklikken van het gewenst testpad in een browser. Voor het inloggen gebruiken we een stub, die automatisch een nieuwe gebruiker aanmaakt. Zo start elke gebruiker in de performancetest met een schone lei en zijn de resultaten voorspelbaar en herhaalbaar. De applicatie is zo opgezet dat alle gebruikers-specifieke zaken (zoals het userId) in de sessie vastliggen. De requestflows, die we doorlopen tijdens de tests, zijn hierdoor voor alle gebruikers hetzelfde. Dat maakt dat we een testpad één keer kunnen opnemen en de opgenomen



Afbeelding 1: globale verdeling van responstijden

requestflow voor een oneindig aantal gebruikers kunnen gebruiken. Er zijn geen ingewikkelde gebruikers-specifieke zaken waar we rekening mee hoeven te houden.

Als we het testpad in de browser doorlopen hebben, exporteren we de requestflow vanuit de browser naar een HAR file. In Chrome kan dit bijvoorbeeld door met de developer tools het netwerkverkeer op te nemen en vervolgens via het contextmenu de HAR met inhoud op te slaan. In Firefox kan dat met een Firebug-plugin. We nemen deze HAR's op in de test resources van een performance module van ons project, zodat ze mee geversioneerd worden met de broncode van het project. Zie de HAR's als een initiële definitie van de requestflow van het performance-testscenario.

GATLING GENEREERT EEN GRAFIEK MET DE GEMIDDELDE RESPONSTIJD PER TEST, ZODAT JE OVER DE LAATSTE 15 TESTS HET VERLOOP VAN DE RESPONSTIJDEN KUNT ZIEN

STATISTICS (Click here to show more) — 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	25800	25787	13	0%	115.878	1	455	738	1370	2081	30436	545	804
GET / Redirect 2	100	100	0	0%	0.449	47	56	68	95	177	240	64	25
GET / Redirect 1	100	100	0	0%	0.449	1	3	3	6	10	14	3	1
GET /	200	200	0	0%	0.898	40	44	46	51	121	163	46	11
GET /dtdl/select	100	100	0	0%	0.449	2	5	7	16	23	25	6	4
GET /dtd...direct 1	100	100	0	0%	0.449	10	29	40	54	66	82	31	13
GET /dtd...direct 2	100	100	0	0%	0.449	29	280	489	766	1017	1319	328	248
GET /dtd....student	100	100	0	0%	0.449	2	4	6	10	21	26	5	3
GET /dtd...direct 3	100	100	0	0%	0.449	2	3	4	7	23	36	3	3
GET /api/course	200	200	0	0%	0.898	11	268	600	1021	1321	1729	372	341
POST /api/state	3800	3800	0	0%	17.067	10	392	609	1038	1515	2417	433	330

Afbeelding 2: details per request

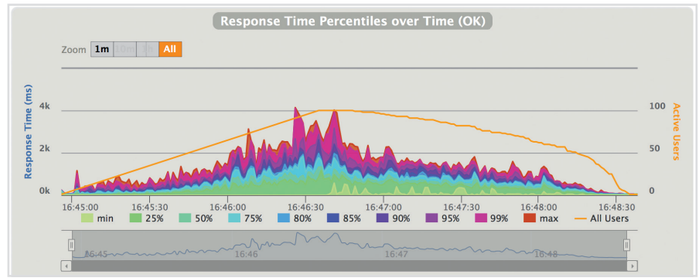
Operationaliseren van de scripts

Voordat de HAR's gebruikt kunnen worden om een performancetest mee uit te voeren, moeten ze eerst worden omgezet naar automatisch uitvoerbare scripts in de Gatling DSL. Dit doen we in twee stappen. De eerste stap is om de HAR file met behulp van de Gatling recorder om te zetten naar de Gatling DSL. We roepen de recorder aan vanuit een eenvoudig Scala-programma. We configureren de recorder hierbij zo, dat deze zaken als requests en responses van statische resources en Google Analytics calls wegfiltert, zodat alleen de requests tussen frontend en backend overblijven. In de tweede stap passen we de gegenereerde DSL aan met een eenvoudig Java-programma. In deze stap brengen we logica in het testscript aan, die normaalgesproken door de frontend wordt uitgevoerd, zoals het uitlezen van een CSRF-token bij inloggen en het zetten van het token bij alle volgende requests. We zorgen er in deze stap ook voor dat alle "think-times" (pauzes tussen requests) een vaste waarde krijgen. Dit is nodig, zodat de resultaten tussen de verschillende versies van de test te vergelijken zijn. Als we dit niet zouden doen, dan zou de timing bij het opnemen van de scripts invloed kunnen hebben op de metingen. Het feit dat deze stappen geautomatiseerd verlopen is essentieel voor het herhalend kunnen opnemen en uitvoeren van tests.

Belangrijk om te weten is dat we de tests hier niet gebruiken om een representatief beeld te krijgen van het maximaal aantal gebruikers dat we in de productieomgeving aankunnen. Deze tests dienen puur als benchmark om een testomgeving onder load te zetten en om de resultaten van verschillende tests met elkaar te kunnen vergelijken om zo een trend en verschillen te ontdekken.

Uitvoeren van de test

Het uitvoeren van de test doen we vanuit Jenkins CI, waarbij we een Maven job aanroepen, die Gatling start en de gegenereerde scripts uitvoert tegen een draaiende testomgeving. De test draait elke ochtend en elke avond.



Afbeelding 3: verloop van responstijden t.o.v. aantal gebruikers

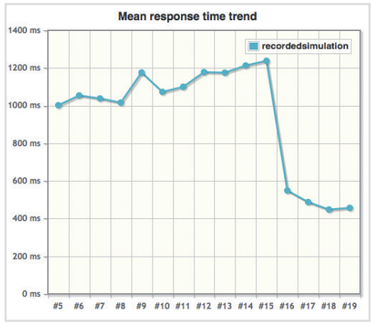
Om de resultaten niet te laten afhangen van het gebruik van de testomgeving gedurende de dag doen we voorafgaand aan elke test een 'warmup' test, die ervoor zorgt dat de applicatie al onder load heeft gestaan. Hierdoor zijn de caches gevuld, de loadbalancers opgewarmd en hebben de JVM's voldoende geheugen gealloceerd.

Verwerken van testresultaten

De resultaten van de test worden in Jenkins opgeslagen door een Gatling-plugin. Gatling genereert een uitgebreide interactieve rapportage (zie **afbeeldingen**) met de details voor een test. De Gatling-plugin genereert ook een grafiek met de gemiddelde responstijd per test, zodat je over de laatste 15 tests het verloop van de responstijden kunt zien. Als je elke dag een test draait, zie je daarmee in één oogopslag hoe de trend van responstijden van je applicatie over de afgelopen twee weken is verlopen. Dit blijkt een erg waardevolle feature, want hiermee hebben we al meerdere potentiële performance-issues ontdekt en kunnen verhelpen.

Conclusie

Het inbedden van performancetests in het ontwikkelproces op hetzelfde niveau als het uitvoeren van unittests en integratietests is eigenlijk een noodzaak in een wereld van Continuous Delivery en DevOps. We hebben hierdoor al meerdere potentiële performance-issues vroegtijdig ontdekt en hierdoor relatief snel en simpel kunnen oplossen. Om dit te kunnen doen, is een goed te automatiseren loadtesting tool noodzakelijk. Gatling leent zich hier goed voor, onder andere door het gebruik van een Scala DSL. Tests zijn eenvoudig op te nemen en de uitvoering is gemakkelijk te automatiseren door de integratie met Jenkins. De rapporten zijn goed bruikbaar en de grafiek met responstijden trend geeft je inzicht in het verloop van de performance van je applicatie over de tijd. Wij zijn in ieder geval enthousiast over Gatling en zijn deze werkwijze inmiddels breder aan het toepassen binnen onze projecten. ■



Afbeelding 4: performancetrend over tests heen in Jenkins

REFERENTIES

[1] http://gatling.io

[2] https://w3c.github.io/web-performance/specs/HAR/Overview.html

[3] http://gatling.io/docs/2.1.7/cheat-sheet.html